

深入浅出 Linux 工具与编程

余国平◎著



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

深入浅出 Linux 工具与编程

余国平◎著

- ◎ 这是一本有思想、有内容的图书
- ◎ 这是一本回答学什么、怎么学的图书
- ◎ 这是一本大学毕业生一次阅读、终生受益的图书
- ◎ 这是一本集实用性、典型性、模仿性案例的图书
- ◎ 这是一本很通俗易懂的图书
- ◎ 这是一本帮你突破技术玻璃纸、一通百通的图书
- ◎ 这是一本包含作者从业十年心得经验的图书
- ◎ 这是一本零起点的Linux专家速成培训教程



电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书系统地论述了 Linux 工具与编程的相关知识。全书内容可分为两部分：Linux 知识的初级部分和高级部分。其中初级部分包括 Linux 操作系统介绍、Linux 命令说明、Linux 常见实用工具（正则表达式、find、sed、awk）、Shell 编程、Linux C 语言程序设计、Linux C 语言开发工具（vi 与 vim 编辑器、gcc、Makefile 和 gdb）；高级部分包括 Linux 进程编程（Linux 进程、Linux 线程、管道与信号、消息队列、信号量和共享内存）、Linux 文件编程、网络编程和 XML 编程。

本书初级部分适合高等院校相关专业学生和 Linux 爱好者学习，高级部分适合 Linux 行业资深从业人员学习。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

深入浅出 Linux 工具与编程 / 余国平著. —北京：电子工业出版社，2011.7
ISBN 978-7-121-13750-1

I. ①深… II. ①余… III. ①Linux 操作系统 IV.①TP316.89

中国版本图书馆 CIP 数据核字（2011）第 104466 号

策划编辑：张春雨

责任编辑：李利健 贾 莉

印 刷：北京天宇星印刷厂

装 订：三河市鹏成印业有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×1092 1/16 印张：35.5 字数：908 千字

印 次：2011 年 7 月第 1 次印刷

印 数：4000 册 定价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zltz@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前言

作者在软件行业从业了十年，先后通过了国家软件水平等级考试的《高级程序员》级别和《系统分析师》级别，参加了大型行业软件如银行核心业务系统、前置系统、数据仓库、金卡工程、银行大小额现代化支付、中间件、支付宝银行端接口等一系列项目设计、开发、测试和实施工作，具有丰富的软件编程经验，同时，一直在多家企业负责新员工的培训工作，对企业员工培训有较多的心得。新员工在大量繁多的技术面前常碰到应该学什么、怎么学的问题；各种技术与工具知识点怎么分优先级和重点问题；技术玻璃纸难以突破，水平难以提高的问题。在企业里，员工怎样在时间有限、精力有限的情况下掌握好实用和有用的技术，满足企业用人的需要？作者总结的经验是培训教材的导向必须注重目的性、思想性、实践性、典型性和实用性，以任务驱动式培训和以目标管理为方法，用图文和言简意赅的语言描述技术思想，用经典程序说明技术思想，用多个项目实战案例解释如何高水平运用技术思想。

模仿是人们快速提高自身能力的捷径，本书的许多程序十分经典，采用实际编程使用的架构，读者可以将编程模板应用于实践。本书的编排注重易学习性、可模仿性和实战性，其中模板化的编程案例和规范化的练习可以让读者在短时间内把书本知识变成自身的能力。本书是一本技术思想深厚的图书，书中的许多内容来源于作者十年技术积累的总结，本书有些技术概念的概括来源于作者多年的思考和感悟，项目案例来源于作者从业的实际项目中。

本书由长期从事一线开发的技术人员编写完成，书中内容通俗易懂，作者力求让 Linux 技术变得简单，读者在阅读本书后能大大增加学习 Linux 技术的信心。本书把繁多的 Linux 技术进行了浓缩，能大大节约读者的学习时间和学习成本。本书注重对技术概念的简要阐述，更注重技术实现，书中对一些技术细节的归纳来源于作者多年工作经验的总结。没有理论，实践是盲目的；没有实践，理论是空洞的。本书力求用言简意赅的理论让读者掌握技术的精髓，用经典程序和项目案例使读者加深对技术理论的理解。本书用精练的概念总结技术，用通俗易懂的语言说明技术，用精心挑选的模板程序和项目案例实现技术。

本书内容

本书分 6 篇，所有的内容注重理论联系实际。每篇的主要内容如下：

第 1 篇 Linux 命令及其工具

本篇包括 Linux 操作系统介绍、Linux 命令说明、Linux 常见实用工具（正则表达式、find、sed、awk）说明及实例练习、Shell 编程语法说明及编程实例。

第 2 篇 Linux C 语言程序设计

本篇包括 C 语言基础、C 语言控制结构、C 语言函数、C 语言数组、结构体及指针、C 语言预编译、格式化 I/O 函数、字符串和内存操作函数、字符类型测试函数、字符串转换函数、Linux C 语言开发工具（vi 与 vim 编辑器、gcc、Makefile 和 gdb）。本篇多次运用堆栈表格对程序运行进行解释，这对于理解计算机语言运行机理非常重要。只有理解的才是最深刻的，理解其运行机理，可以触类旁通、一通百通，移植到理解 C++ 语言和 Java 等语言。

第 3 篇 Linux 进程

本篇包括 Linux 编程基本概念、Linux 进程、Linux 线程、管道与信号、消息队列、信号量和共享内存。Linux 进程章节中守护进程模板和数据仓库多进程处理案例可以应用到实际项目中。本篇 Linux 进程间通信程序范例是实际项目中精简的 Demo 程序，程序模型和使用方法与实际项目中类似。

第 4 篇 Linux 文件

本篇包括 Linux 文件编程，该部分内容对文件函数进行了分类总结，并提供了典型范例。

第 5 篇 网络编程

本篇包括网络知识基础、Socket 编程。Socket 编程章节包括 TCP 并发服务器案例、TCP 迭代服务器案例、文件服务器案例、UDP 服务器编程、UDP 广播、UDP 多播、UNIX/Linux 域套接字编程等。

第 6 篇 XML 编程

本篇包括 XML 概念、XML 语法、XPath 语法、libxml 编程、支付宝银行端接口 XML 项目案例。本篇内容是目前市面上对 Linux 下 XML 编程总结非常全面的，在实际项目开发中有较大的借鉴意义。

本书特色

1. 零起点的企业级培训教程

读者只需具有大学计算机专业及相关专业一般水平，即可对本书进行阅读和练习。本书内容

深入浅出 Linux 工具与编程

通俗易懂，图文并茂，注重知识点的总结概括和分类。知识结构注重层层递进，以达到让读者在低起点向专家迈进的目的。有 Linux 从业基础部分（如 Linux C 语言程序设计），有 Linux 从业素质能力培养部分（如 Linux 命令及其工具、Linux C 语言开发工具），也有 Linux 编程专家水平能力训练部分（如 Linux 进程编程、Linux 文件编程、Linux 进程间通信、网络编程与 XML 编程）。本书涵盖了 Linux 原理篇、命令篇、工具篇和程序篇。

专业就是“简单的事情”重复做，做到专业，就是把复杂的事情简单化，其方法为分类、分层、总结、模板化和流程化。而本书正是致力于这一目的，把复杂的技术简单明了地呈现在读者面前，帮助读者成为专业人士。

2. 大量的企业级实训内容

本书的许多章节是作者关于企业级培训的实训内容，知识点注重目标明确、言简意赅、分清主次、项目导向，以求达到简洁不简单效果。本书属于企业级实训教程，以 Linux 行业从业素质能力培养为导向，以实际应用为目标，以简洁的理论和经典练习为过程，以期达到快速提高读者的职业水平和职业能力。本书采用 Linux 行业素质能力模型的训练方法，即将 Linux 从业知识点逐条列出，并把知识点整合到规范化练习案例中，以达到让读者通过模仿练习快速把知识变为能力的目的。如本书 Linux 工具与命令章节，读者只要按照练习，即可达到 Linux 行业从业所需的中级水平，而 Shell 章节按照练习即可快速提高到 Linux 行业老员工的水平。本书这些企业级培训内容能帮助读者在短时间内学到实用且够用的 Linux 开发知识。

3. 学什么，怎么学

一门技术是很难在短时间内学好的，但通常可以快速学会常用和关键的技术。本书以实用论为导向，丢弃了项目开发中用不到的众多技术细节。

本书注重理论联系实际，作者把自己十年的项目经验整合到本书中，将 Linux 从业的知识进行分类、总结，并辅以案例讲解，许多知识点都以实际工作所需知识为准，也是以作者所掌握的主要和重点知识为准。

书中的许多章节配有典型程序和规范化案例练习，学完理论后按照案例练习，即可达到技术的提升。本书内容的选取完全参照作者十年从业经验所用到的知识，言简意赅的图文讲解和规范化案例练习告诉读者怎么学。

4. 多个实用项目案例

本书包含多个经典的项目案例，如 Shell 章节的备份脚本、C 语言章节的实用日志库、Linux 进程章节数据仓库多进程案例、网络章节的实用文件服务器和实用通信库、XML 章节支付宝银行端接口项目的 XPath 库。这些案例具有较大的实用参考价值。

5. XML 章节填补市场空白

XML 是软件行业经常使用的技术，经常应用在数据交换、Web 服务、内容管理、电子商务、配置脚本等方面。目前市面上的图书缺少针对 XML 开发技术的案例介绍，本书对这部分内容进行了专门的总结，同时提供了丰富的练习和经典的项目案例。

6. Linux 专家速成培训教程

时间是人类发展的空间，赢得时间就是赢得个人发展的空间。在个人的职业生涯中，一步领先常常可以做到步步领先。读者只要静下心来用一个月时间阅读本书，并进行练习，就可以大大提高 Linux 编程技术水平，如果完全掌握本书内容，即可达到 Linux 专家水平。可以说，本书是一本通向 Linux 专家之路的速成教程。

由于作者水平有限，书中错漏之处在所难免，恳请读者批评指正。

目录

第 1 篇 Linux 命令及其工具	
第 1 章 Linux 系统与命令	2
1.1 Linux 操作系统	2
1.1.1 Linux 重要概念	2
1.1.2 Linux 组成	3
1.1.3 Linux 目录结构	3
1.1.4 Linux 操作系统的组成	5
1.1.5 Linux 用户管理	6
1.1.6 Linux 文件管理	6
1.2 Linux 命令	7
1.2.1 Linux 命令帮助	7
1.2.2 Linux 命令的符号及意义	8
1.2.3 Linux 命令	9
第 2 章 Linux 常用实用工具	18
2.1 正则表达式	18
2.2 find 查找命令	20
2.2.1 find 语法	20
2.2.2 find 实例练习	22
2.3 sed	24
2.3.1 sed 语法	24
2.3.2 sed 实例练习	26
2.4 awk	29
2.4.1 awk 语法	29
2.4.2 awk 实例练习	36
第 3 章 Shell 编程	40
3.1 Shell 环境变量	40
3.1.1 环境变量说明	40
3.1.2 用户常用的系统环境变量	41
3.1.3 用户登录脚本示例	42
3.2 Shell 的符号、变量及运行	43
3.2.1 Shell 中的符号及其含义	43
3.2.2 “反引号”命令替换	44
3.2.3 Shell 变量	45
3.2.4 Shell 脚本执行	50
3.2.5 Shell 退出状态	50
3.3 Shell 的输入和输出	51
3.3.1 Shell 的输入	51
3.3.2 Shell 的输出	52
3.4 Shell 测试条件	53
3.5 Shell 的流程控制结构	57
3.5.1 if 语句	57
3.5.2 case 语句	59
3.5.3 while 语句	60
3.5.4 until 语句	61
3.5.5 for 语句	62
3.5.6 跳转语句	64
3.6 Shell 数组	64

3.7	Shell 函数	65	6.3.1	指针概念	134
3.8	I/O 重定向	67	6.3.2	sizeof、void、const 说明	139
3.9	Shell 内置命令	68	6.3.3	指针变量作为函数参数	140
3.10	实用 Shell 脚本	73	6.3.4	指针的运算	142
 第 2 篇 Linux C 语言程序设计			6.3.5	指向数组的指针变量	144
 第 4 章 C 语言基础			6.3.6	数组名作为函数参数	146
4.1	C 语言基本概念	76	6.3.7	函数指针变量	148
4.2	常量与变量	84	6.3.8	返回指针类型函数	149
4.3	运算符	89	6.3.9	指向指针的指针	150
4.4	C 语言控制结构	96	6.3.10	结构指针	150
4.4.1	if 语句	97	6.3.11	动态存储分配	152
4.4.2	switch 语句	100	6.3.12	指针链表	153
4.4.3	goto 语句	101	6.3.13	指针数据类型小结	154
4.4.4	while 语句	102	 第 7 章 C 语言预处理		156
4.4.5	do-while 语句	103	7.1	define 宏定义	156
4.4.6	for 语句	104	7.2	typedef 重定义	157
4.4.7	break 和 continue 语句	106	7.3	inline 关键字	158
 第 5 章 C 语言函数			7.4	条件编译	158
5.1	函数简述	107	7.5	头文件的使用	159
5.2	函数变量	110	 第 8 章 格式化 I/O 函数		161
5.3	函数定义与调用	110	8.1	格式化输出函数	161
5.3.1	函数定义	110	8.1.1	输出函数原型	161
5.3.2	函数的参数与返回值	111	8.1.2	输出函数格式说明	162
5.3.3	函数调用	115	8.2	格式化输入函数	165
 第 6 章 C 语言数组、结构体及指针			8.2.1	输入函数原型	165
6.1	C 语言数组	119	8.2.2	输入函数格式说明	165
6.1.1	数组概述	119	 第 9 章 字符串和内存操作函数		169
6.1.2	一维数组	120	9.1	字符串操作函数说明	169
6.1.3	二维数组	124	9.2	字符串函数操作	170
6.1.4	字符数组	127	9.3	字符类型测试函数	179
6.1.5	冒泡法排序	128	9.4	字符串转换函数	180
6.2	C 语言结构	129	 第 10 章 标准 I/O 文件编程		182
6.2.1	结构概念	129	10.1	文件打开方式	183
6.2.2	结构变量	131	10.2	标准 I/O 函数说明及程序范例	185
6.3	指针	134	 第 11 章 Linux C 语言开发工具		200
			11.1	vi 与 vim	200

11.1.1	vi 与 vim 概述	200
11.1.2	指令模式	201
11.1.3	末行模式	207
11.1.4	vim 个人使用经验	210
11.1.5	vim 的使用	211
11.1.6	文件编码	214
11.1.7	vi 与 vim 模拟练习	217
11.2	gcc	218
11.2.1	gcc 简要说明	218
11.2.2	gcc 参数	220
11.3	Makefile	224
11.3.1	Makefile 简介	225
11.3.2	Makefile 语法	227
11.3.3	Makefile 的运行	231
11.3.4	Makefile 的扩展话题	232
11.4	gdb	233
11.4.1	gdb 语法	233
11.4.2	gdb 调试	238

第 3 篇 Linux 进程

第 12 章	Linux 进程编程	242
12.1	Linux 进程编程基本概念	242
12.1.1	登录	242
12.1.2	文件和目录	243
12.1.3	输入和输出	243
12.1.4	程序与进程	244
12.1.5	ANSI C	245
12.1.6	用户标识	247
12.1.7	出错处理	247
12.1.8	Linux 信号、时间值与系统调用	249
12.2	Linux 进程环境	256
12.3	Linux 进程控制	267
12.4	进程关系	289
12.5	守护进程与多进程并发案例	293
12.5.1	守护进程的编写	293
12.5.2	多进程并发项目案例	296

第 13 章	Linux 线程编程	300
13.1	线程简要说明	300
13.2	线程主要函数	302
13.3	线程编程	308
13.3.1	线程创建	308
13.3.2	终止线程	310
13.3.3	线程互斥	312
13.3.4	线程同步	315

第 14 章	Linux 进程间通信——管道与信号	318
14.1	进程间通信概述	318
14.2	管道	319
14.2.1	pipe 管道	320
14.2.2	标准流管道	324
14.2.3	命名管道（FIFO）	325
14.3	信号	328
14.3.1	信号概述	328
14.3.2	信号的发送和捕捉函数	332
14.3.3	信号的处理	337

第 15 章	System V 进程间通信	346
15.1	System V 进程间通信的键值	346
15.2	消息队列	350
15.2.1	消息队列简要说明	351
15.2.2	消息队列函数	352
15.2.3	消息队列使用程序范例	355
15.3	信号量	360
15.3.1	信号量简要说明	360
15.3.2	信号量函数	361
15.3.3	信号量应用程序示例	364
15.4	共享内存	366
15.4.1	共享内存简要说明	366
15.4.2	共享内存函数	368
15.4.3	共享内存应用范例	370

第 4 篇 Linux 文件

第 16 章	Linux 文件编程	376
16.1	文件系统函数	376

16.2	初级文件 I/O 函数	392	
16.3	标准 I/O 的缓冲和刷新	399	
 第 5 篇 网络编程			
 第 17 章 网络知识基础			402
17.1	网络体系结构及协议	402	
17.1.1	网络体系结构概念	402	
17.1.2	TCP/IP 模型	405	
17.1.3	网络分类与广域网	407	
17.1.4	网络地址	410	
17.2	TCP/IP 协议簇报文格式	412	
 第 18 章 socket 编程			416
18.1	套接字说明及函数说明	416	
18.1.1	套接字说明	416	
18.1.2	socket 地址说明及 转换函数	419	
18.1.3	socket 主要函数说明	424	
18.2	TCP 套接字编程	432	
18.2.1	TCP 套接字编程模型	432	
18.2.2	迭代服务器编程	436	
18.2.3	并发服务器编程	437	
18.3	TCP 文件服务器项目案例	443	
18.4	UDP 编程	458	
18.4.1	普通 UDP 服务器编程	458	
18.4.2	UDP 广播	461	
18.4.3	UDP 多播	464	
18.5	原始套接字	469	
18.5.1	原始套接字说明	469	
18.5.2	原始套接字举例	471	

18.6	本地进程间套接字	478	
18.6.1	非命名 UNIX 域套 接字管道	478	
18.6.2	UNIX 域套接字	479	
18.7	I/O 编程模型	483	
 第 6 篇 XML 编程			
 第 19 章 XML 概念与语法			490
19.1	XML 概述	490	
19.2	XML 语法	493	
19.3	XPath 语法	503	
19.3.1	XPath 基本语法	503	
19.3.2	XPath 位置路径	508	
19.3.3	XPath 示例	514	
 第 20 章 libxml 编程			518
20.1	libxml 编程基础	518	
20.1.1	libxml 的安装	518	
20.1.2	libxml 主要的数据类型	519	
20.1.3	libxml 的主要函数说明	522	
20.1.4	XML 常见操作	530	
20.2	libxml 高级编程进阶	536	
20.2.1	理解 DOM 树	536	
20.2.2	libxml 编程实例练习	541	
20.2.3	支付宝银行端接口 XML 项目案例	547	
 附录			552
 参考文献			555

第 1 篇

Linux 命令及其工具

- ❖ 第 1 章 Linux 系统与命令
- ❖ 第 2 章 Linux 常用实用工具（正则表达式、find、sed、awk）
- ❖ 第 3 章 Shell 编程

学海聆听：

- 合抱之木，生于毫末；九层之台，起于垒土；千里之行，始于足下。
- 学而不思则罔，思而不学则殆。
- 知识来源于学习、思考、顿悟、实践和交流。
- 非学无以广才，非志无以成学。
- 临渊羡鱼，不如退而结网。
- 纸上得来终觉浅，绝知此事要躬行。
- 在模仿中成长，在创新中成功。
- 好的开始等于成功的一半。
- 梦想引领未来，行动成就精彩。
- 确定目标—支配行动—克服困难—实现目标。
- 工欲善其事，必先利其器。
- 博学之，审问之，慎思之，明辨之，笃行之。

第 1 章

Linux 系统与命令

本章分两部分：Linux 操作系统和 Linux 命令。Linux 操作系统部分简要介绍 Linux 操作系统的概念、组成和实现。Linux 命令部分用表格列出了 Linux 常见命令和 Linux 次常见命令，简要地说明了其使用方法。熟练地掌握 Linux 命令和工具是非常必要的，能提高工作效率。

1.1 Linux 操作系统

1.1.1 Linux 重要概念

Linux 操作系统有如下一些重要的概念。

① 机器指令：CPU 可识别的二进制指令集，许多指令可以与汇编指令一一对应。主要分为四类：计算指令、存数据指令、取数据指令、中断指令。

② 程序：用计算机语言编写的源代码叫做源程序，经过编译程序编译后形成的可执行码叫做可执行程序，程序是静态的。

③ 进程：程序的一次执行过程叫做进程。进程是动态的，其堆栈空间存在于内存中，CPU 从内存中取程序执行代码执行相关操作。一个程序执行一次，就产生一个进程，所以，程序与进程是一对多的关系。

④ 中断：顾名思义，即中止和打断。中断是操作系统中止当前运行程序去执行相应中断号程序的过程，执行完成中断号程序后，回到原程序断点并往下继续执行。

⑤ 文件系统：组织、存储、检索和管理文件的系统。

⑥ 文件：一个具有符号的一组相关联元素的有序序列。

⑦ 文件名：即文件的名称，一个目录下的文件名不能重复。

⑧ 目录：为了方便管理文件，就形成了树形目录结构。目录下可以有文件或目录，目录类似于 Windows 系统下的文件夹。

⑨ 相对路径：相对当前路径的路径。如当前目录为 /home/test/hello，到达 /home/

test/hello/why 目录，可通过相对路径方式 `cd ./why` 到达。

⑩ 绝对路径：从根目录（/）开始的路径。如当前目录为 `/home/test/hello`，若要到达 `/home/test/hello/why` 目录，可通过绝对路径方式 `cd /home/test/hello/why` 实现。

⑪ Shell：Shell 是一个交互性命令解释器，Linux 用户可以通过编写 Shell 程序完成一些管理工作。

⑫ inode 节点：文件系统的基本对象。每个文件和目录对应一个 inode，inode 在一个分区内是唯一的，它可以是一个正常文件、一个目录、一个符号链接，或者是其他内容。

1.1.2 Linux 组成

Linux 有四个主要部分，分别为 Shell、文件系统、内核、实用工具。

Linux 内核主要由五个子系统组成，分别为进程调度、内存管理、虚拟文件系统、网络接口、进程间通信。

Linux 系统管理包括用户管理、文件管理、设备管理、进程管理、网络管理、作业管理和实用工具。Linux 实用工具包括压缩/解压工具、Shell 工具等。

1.1.3 Linux 目录结构

1. 目录结构

Linux 文件系统是树状目录层次结构。/为根目录，根目录为一级子目录，各子目录名有特定的含义，这些目录存放和管理特定的文件。Linux 目录结构如图 1-1 所示。

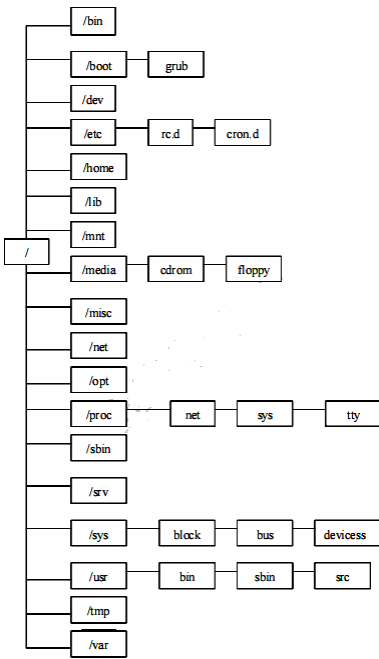


图 1-1 Linux 目录结构

2. 目录说明

为了方便文件的管理，Linux 操作系统建有许多一级子目录或多级子目录，这些目录存放和管理特定的某一类文件。Linux 目录结构说明如表 1-1 所示。

表 1-1 Linux 目录结构说明

目录名称	说 明
/bin	bin 就是二进制 (binary) 英文缩写。该目录下存放 Linux 常用操作命令的执行文件，如 mv、ls、mkdir 等。有时，这个目录的内容和/usr/bin 中的内容一样，它们都是放置一般用户使用的执行文件
/boot	这个目录下存放着操作系统启动时所要用到的程序
/dev	该目录包含了 Linux 系统中使用的所有外部设备。要注意的是，这里存放的并不是外部设备的驱动程序，它实际上是一个访问这些外部设备的端口。由于在 Linux 中，所有的设备都当做文件进行操作，比如：/dev/cdrom 代表光驱，用户可以非常方便地像访问文件、目录一样对其进行访问
/etc	该目录下存放了系统管理时要用到的各种配置文件和子目录，如网络配置文件、文件系统、系统配置文件、设备配置信息设置用户信息等。系统在启动过程中需要读取其参数进行相应的配置
/etc/rc.d	该目录主要存放 Linux 启动和关闭时要用到的脚本文件
/etc/rc.d/init	该目录存放 Linux 服务默认的所有启动脚本。在新版本的 Linux 中还用到/etc/xinetd.d 目录下的内容
/home	该目录是 Linux 系统中默认的用户工作根目录。如果建立一个名为“xx”的用户，那么在/home 目录下就有一个对应的“/home/xx”路径用来存放该用户的主目录
/lib	该目录是用来存放系统动态链接共享库的。几乎所有的应用程序都会用到这个目录下的共享库。因此，千万不要轻易对这个目录进行任何操作
/lost+found	该目录在大多数情况下都是空的。只有当系统产生异常时，才会将一些遗失的片段放在此目录下
/media	该目录下以前是光驱和软驱的挂载点，现在 Linux 已经可以自动挂载光驱和软驱了
/misc	该目录下存放从 DOS 下进行安装的实用工具，一般为空
/mnt	该目录是软驱、光驱、硬盘的挂载点，也可以将别的文件系统临时挂载到此目录下
/proc	该目录是用于放置系统核心与执行程序所需的一些信息，这些信息是在内存中由系统产生的，故不占用硬盘空间
/root	该目录是超级用户登录时的主目录
/sbin	该目录用来存放系统管理员常用的系统管理程序
/tmp	该目录用来存放不同程序执行时产生的临时文件。Linux 安装软件的默认安装路径通常位于该目录下
/usr	这是一个非常重要的目录，用户的很多应用程序和文件都存放在这个目录下，类似于 Windows 下的 Program Files 目录
/usr/bin	系统用户使用的应用程序
/usr/sbin	超级用户使用的比较高级的管理程序和系统守护程序
/usr/src	内核源代码默认的放置目录
/srv	该目录存放一些服务启动之后需要提取的数据
/sys	这是 Linux 2.6 内核的一个很大的变化。该目录下安装了 Linux 2.6 内核中新出现的一个文件系统-sysfs 文件系统，此系统集成了三种文件系统的信息：针对进程信息的 proc 文件系统、针对设备的 devfs 文件系统以及针对伪终端的 devpts 文件系统。/sys 目录是内核设备树的一个直观反映。当一个内核对象被创建的时候，对应的文件和目录也在内核对象子系统中被创建
/var	这也是一个非常重要的目录，很多服务的日志信息都存放在这里

1.1.4 Linux 操作系统的组成

Linux 操作系统的层次结构如图 1-2 所示。用户应用程序如办公软件，或者自己编写的应用程序服务；操作系统服务指 Shell 解释器、系统调用接口等；操作系统内核是操作系统在内核态完成系统功能的程序；硬件系统指硬件设备驱动程序。

Linux 内核系统模块主要由五个模块构成，它们分别是：进程调度模块、内存管理模块、文件系统模块、进程间通信模块和网络接口模块。它们的相互依赖关系如图 1-3 所示，由图可以看出，所有的模块都与进程调度模块存在依赖关系，因为它们都需要依靠进程调度程序来挂起（暂停）或重新运行它们的进程。通常，一个模块会在等待硬件操作期间被挂起，而在操作系统完成硬件操作后才可继续运行。

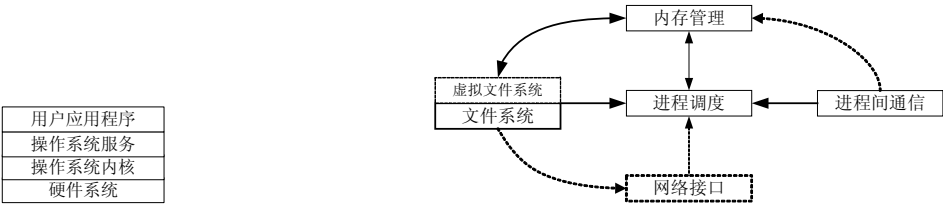


图 1-2 Linux 操作系统的层次结构图

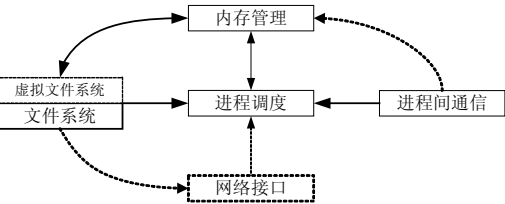


图 1-3 Linux 内核系统模块结构及相互依赖关系图

Linux操作系统属于系统软件，软件都是建立在硬件基础上去实现相应的功能。Linux内核结构图如 1-4 所示，此图划分了三个层次，分别为硬件级、内核级、用户级。硬件级指的是电脑硬件，如CPU、显卡、显示器等。内核级指的是操作系统内核功能（除硬件控制部分），它处于操作系统的核心态，内核代码可以无限制地对系统存储、外部设备进行访问。图中内核级的硬件控制一般指的是BIOS（基本输入/输出系统）程序，BIOS的主要功能是为计算机提供最底层的、最直接的硬件设置和控制。用户级是操作系统提供给用户的调用接口，它处于操作系统的用户态，应用程序要访问系统硬件资源需要通过系统调用去实现，应用程序调用系统调用时，用户进程将由用户态切换到核心态。

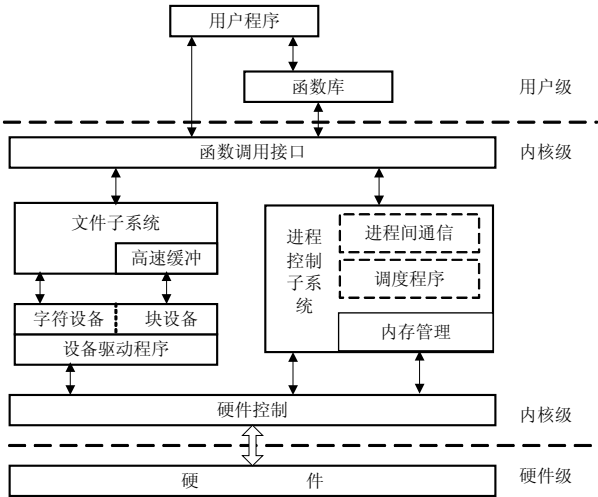


图 1-4 Linux 内核结构图

1.1.5 Linux 用户管理

Linux 系统是一个多用户系统，能做到不同的用户同时访问不同的文件。因此，一定要有文件权限控制机制。

Linux 中每一个用户有一个唯一的用户 ID。Linux 下的用户分为三类：超级用户、系统用户和普通用户。

超级用户的用户名为 `root`，它拥有一切权限。

系统用户是 Linux 系统正常工作所必需的内建用户，主要是为了满足相应的系统进程对文件属主的要求而建立的，系统用户不能用来登录，如 `bin`、`daemon`、`adm`、`lp` 等用户。

普通用户是为了让使用者能够使用 Linux 系统资源而建立的，大多数用户属于此类。

一个用户属于一个用户组，也可以属于多个用户组，Linux 中每一个组有唯一的组 ID。

在 Linux 操作系统中，超级用户（`root`）的权限是最高的，也被称为超级权限的拥有者。普通用户无法执行的操作，`root` 用户都能完成，所以也被称为超级管理用户。在系统中，每个文件、目录和进程都归属于某一个用户，没有用户许可，其他普通用户是无法操作的，但对 `root` 除外。

环境变量指该用户环境中定义生效的变量。用户环境变量定义在 `.profile` 和 `.bash_profile` 文件中，支持递归定义。用户可以在用户环境下用 `env` 命令查看环境变量。

1.1.6 Linux 文件管理

Linux 设计哲学继承了 UNIX 血统，一切皆为文件。Linux 文件包括如下类型：

- `d`: 目录文件，目录也是一种文件（`directory`）。
- `l`: 符号链接（指向另一个文件）（`link`）。
- `b`: 块设备文件（`block device`）。
- `c`: 字符设备文件（`character device`）。
- `p`: 命名管道文件（`named pipe`）。
- `s`: 套接字文件（`socket`）。
- `-`: 普通文件，或者更准确地说，不属于以上几种类型的文件。

Linux 中的文件属性如图 1-5 所示。文件属性通常有 10 位，第 1 位为该文件类型，第 2~4 位为该文件用户的权限，第 5~7 位为该组用户的权限，第 8~10 位是其他用户权限。

首先，Linux 中的文件按用户权限可分为三个不同的用户级别：文件拥有者（`u`）、所属的用户组（`g`）和系统里的其他用户（`o`）。每类用户级别对文件又可以定义三种不同的访问权限：可读（`r`）、可写（`w`）和可执行（`x`）。具体说明如下。

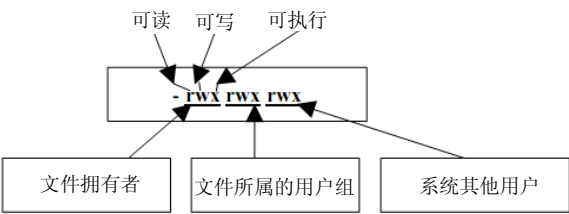


图 1-5 Linux 文件属性图

① 用户权限的三个分类：

- 文件属主 (user)：创建该文件的用户。
- 同组用户 (group)：该文件用户组中的任何用户，同组用户又称为属组。
- 其他用户 (other)：即不属于拥有该文件用户组的任何用户。

② 文件访问权限的三个分类：

- 读 (r)：可以显示该文件的内容。
- 写 (w)：可以编辑或删除它。
- 执行 (x)：如果该文件是一个 Shell 脚本或程序，则表示可以执行。

文件访问权限由三位二进制数表示，0 表示无此权限，1 表示有此权限。例如，r-x 表示对该文件有读和执行权限，用二进制数表示为 101。用 `ls -l` 命令可以查看每一个文件权限和类型。

举例说明，某文件的文件属性为 `-rwxrw-r--`，表示该文件为普通文件，文件属主用户对该文件可读、可写、可执行，同组用户对该文件可读、可写，而其他用户只有对文件的可读权限。

1.2 Linux 命令

Linux 命令完成对 Linux 系统的管理，是在 Linux 从业中经常需要使用到的一项基本技能。Linux 命令虽然有很多，但经常使用的却不多，而且在 Linux 中许多命令参数的含义是相同或相似的。下面只列出 Linux 常用命令和次常用命令。

1.2.1 Linux 命令帮助

Linux 命令帮助可通过 `help` 和 `man` 两个命令获得。利用 `help` 可以获得简洁的帮助信息，以及 Linux 命令的使用方法和参数。而 `man` 帮助命令可以获得 Linux 命令的详细说明。

1. help 帮助命令

`help` 命令使用格式为：

Linux 命令 `--help`

例：`cat --help`

`mkdir --help`

2. man 帮助命令

man 是 Linux 的详细帮助命令，为 Linux 学习者提供了详细的帮助说明。由于 Linux 命令和函数众多，Linux 将帮助信息分为九大类，如表 1-2 所示。

表 1-2Linux 命令分类

类型序号	类型名称	说 明
1	命令	普通用户命令
2	系统调用	内核接口
3	函数库调用	普通函数库中的函数
4	特殊文件	/dev 目录中的特殊文件
5	文件格式和约定	/etc/passwd 等文件的格式
6	游戏	游戏帮助信息
7	杂项和约定	标准文件系统布局、手册页结构等杂项内容
8	系统管理命令	系统管理命令帮助信息
9	内核例程	提供 Linux 内核开发的帮助

Linux 帮助命令格式为：

```
man 类型 命令名称或函数名称
```

其中类型可省略，但名称重复时，类型不能省略。例如，write 在 Linux 中既为命令名称，又为系统调用名称，此时需要接类型确定查找的是命令还是系统调用。

查找命令帮助方法为：

```
man pwd
```

等同于

```
man 1 pwd。
```

查找系统调用帮助方法为：

```
man 2 write。
```

1.2.2Linux 命令的符号及意义

Linux 命令行下常见的符号及含义如下：

- | ：管道。例如，ps -ef|grep user1 是将 ps -ef 的输出作为 grep user1 的输入。
-) ：覆盖输出。例如，cat aa >cc 是将 aa 的内容输入到 cc 中，cc 文件原有内容会全部丢失。
-) ：创建空文件。例如，) dd 是创建名称为 dd 的空文件。
- < ：输入。
- >> ：追加输出。
- 0 ：标准输入，相当于 stdin，对应键盘。

- 1：标准输出，相当于 stdout，对应屏幕。
- 2：标准错误，相当于 stderr，对应屏幕。
- &：后台命令。
- “ ”：双引号表示除\、\$、'和"外，由双引号引起来的字符为普通字符。
- ‘ ’：单引号引起来的字符均作为普通字符。
- ``：命令替换。反引号引起来的字符串作为Shell命令执行。
- ~：表示主目录。
- []：表示括号中的内容为可选。

1.2.3 Linux 命令

1. 用户管理说明

新建、修改、删除用户和组可以在图形界面下操作，命令行命令只作了解即可。

2. Linux 常用命令

Linux 中常用的命令如表 1-3 所示，其中的“空”表示不带参数。

表 1-3 Linux 常用命令

命 令	参数及说明	例 子	附加说明
1. 文件及目录管理			
ls 显示目录	空：不显示隐含文件 -a：显示所有的文件 -l：显示详细列表	ls ls -a ls -l	不显示隐含文件 显示文件包含隐含文件。隐含文件指以.开头的文件 显示文件的详细列表
chgrp 设置文件组 (root 权限)	空：对该文件设置文件组 -R：递归对目录下所有的文件设置文件组	chgrp staff /u chgrp -R staff /u	将/u 的属组更改为 staff 将/u 及其子目录下所有文件的属组更改为 staff
chown 设置文件用户 (root 权限)	空：对该文件设置文件组 -R：递归对目录下所有的文件设置文件组	chown root /u chown root:staff /u chown -R root /u	将/u 的属主更改为 root 将/u 属主改为用户 root，同时也将其属组更改为 staff 将/u 及其子目录下所有文件的属主更改为 root
chmod 设置文件权限 (有两种设置方式： 数字方式设置和字母方式设置)	方式一：[-R] 数字 文件名 方式二： [-R][u,g,o,a][+-][r,w,x] 其中： -R：表示递归；u：表示该用户； g：表示组；o：表示其他；a：表示所有； [+-]表示增加或减少； [r,w,x]表示读、写、执行权限	chmod 752 aa chmod u+x aa chmod -R 752 a	把 aa 的权限变为-rwxr-x-w- 在 aa 的用户权限上加 x 权限 将 a 目录下所有文件的权限改为 752

续表

命 令	参数及说明	例 子	附加说明
pwd 查看用户路径	显示当前用户目录路径	pwd	显示当前用户目录路径
cd 改变当前路径	空: 回到该用户主目录 cd: 路径	cd cd /home/usr1/aa cd ..	回到该用户主目录 到路径/home/usr1/aa 目录下 到该目录的上级目录下
mkdir 建立目录	空: 存在该目录时报错 -p: 有目录时不报错	mkdir ccc mkdir -p ccc	创建 ccc 目录 创建 ccc 目录, 存在 ccc 目录时不报错
rmdir 删除目录	空: 只有目录为空时才能删除目录 -p: 递归删除下级目录	rmdir ccc rmdir -p a/f/x	删除 ccc 目录 删除 a/f/x、a/f、a 三个目录
rm 删除文件	-f, --force: 强制删除 -i: 在要删除前需要确认 -r, -R: 递归删除目录及其内容	rm xxx rm -i * rm -rf ccc	删除文件 xxx 在提示情况下删除当前目录下的所有文件 递归删除 ccc 目录下所有的文件和目录, 而且不提示
cp 复制文件	空 -i: 在要复制前需要确认 -r, -R: 递归复制目录及其内容	cp aa cc cp -i cc dd cp -r xx yy	复制 aa 文件到 cc 文件中 复制 aa 到 dd 时会提示确认 递归将 xx 目录复制到 yy 目录
mv 移动文件	空 -i: 更名前需确认 -f: 更名前不需确认	mv aa cc	将 aa 文件移到 cc 文件
cat 显示文件	空 -n: 显示文件前加行号	cat file cat -n file	将 file 显示到屏幕 显示到屏幕, 并在每行前加行号
more 分屏显示文件	分屏显示内容	more file	分屏显示 file
head 显示文件头	空: 显示文件前 10 行 -n: 显示文件前 n 行	head file head -n 30 file	显示 file 前 10 行 显示 file 前 30 行
tail 显示文件尾	空: 显示文件尾 10 行 -n: 显示文件尾 n 行 -f: 实时追加显示文件	tail file tail -n 30 file tail -f file	显示 file 尾 10 行 显示 file 尾 30 行 追加显示文件
touch 改变文件时间	改变文件时间	touch xxx	改变 xxx 文件的时间
diff 比较文件	diff 文件 1 文件 2	diff file1 file2	比较两个文件的不同
file 查看文件类型	空: 查看文件类型	file file1	查看 file1 文件类型
sort 排序	-n: 按数字 -u: 去重 -k: 列数 -r: 反序 -t: 分隔符 -o: 输出到文件	sort file sort -n file sort -k 3 file sort file -o file1 sort -n -k 2 -t : fil	对 file 文件按第一列进行排序 对 file 按数字排序 对 file 按第 3 列排序 将 file 的排序结果输出到 file1 对 fil 按分隔符: 对第 2 列进行数字排序

续表

命 令	参数及说明	例 子	附加说明
uniq 去掉重复行		uniq file>file1	将 file 的重复行去掉，并把结果输出到 file1 中
wc 文本统计	空：统计下述三种数字 -l：统计行数 -c：统计字节数 -m：统计单词数	wc aaa wc -l aaa	对 aaa 文件进行统计 统计 aaa 文件的行数
grep 在文本中搜索单词	grep：'单词' 文件 -i：不区分大小写 -n：显示行号 -v：显示不包含匹配模式的行	grep 'hel to' aaa grep -i AA aaa grep -v AA aaa	从 aaa 文件中搜索 hel to 的单词行 搜索 aa、AA、aA、Aa 的行 不显示 AA 的行，其他行显示
2. 系统管理与设置			
ps 查找进程	-e：列出所有的进程 -f：显示 UID、PPID、C 与 STIME 的栏位	ps -ef grep proc ps -ef grep user1	查找 proc 的进程信息 查找用户 user1 的进程信息
kill 发送信号	kill -l [signal] kill [-s signal -p] [-a] pid	kill -l kill -9 进程号 kill -9 -1 kill 1234	列出所有信号的名称 杀死指定进程号的进程 杀死系统中所有的进程 向进程号为 1234 的进程发送 SIGTERM（终止）信号
passwd 修改用户密码	passwd passwd 用户名	passwd 回车 passwd user1	修改本用户密码 修改 user1 用户密码，需 root 用户权限
env 查看环境变量	空	env	查看该用户下的环境变量
su 改变用户	su 用户名 su - 用户名	su user1 su - user2	不改变用户环境变量切换到 user1 环境 改变用户环境变量切换到 user2 环境 使用 env 命令查看两者的不同
export 对环境变量进行输出	export 变量 export 变量=值	AA=cc; export AA export AA=cc	将 AA 变量赋值为 cc 并输出 同上，只不过是两种定义方式而已。 一般在 .profile 文件中定义
echo 输出变量或文本	echo \$变量 echo "文本"	echo \$HOME echo "hello hello"	输出 HOME 变量的值 输出文本 hello hello 到屏幕上
umask 设置用户文件掩码位	umask 显示掩码位 umask -S 设置掩码位 (当 umask 为 0022 时,新创立的文件权限为 0777&~0022, 即 0755)	umask umask -S 0027 >aa, ls -l aa	显示该用户的掩码位 设置该用户掩码位为 0027 创建 aa 用户，查看 aa 文件权限是否为 0750
clear 清屏	空	clear	清屏
date 查看或设置时间	date 查看时间 date -s 设置时间	date date -s 2009/07/27 date -s 10:25:36	查看时间 设置日期（需 root 权限） 设置时间（需 root 权限）

续表

命 令	参数及说明	例 子	附加说明
df 查看磁盘使用情况	空: 默认使用 -a 参数 -i: 显示 inode 节点情况 -a: 显示所有信息	df df -i	显示磁盘使用情况 显示 inode 节点使用情况
who 显示登录用户	空	who	显示登录用户
uname	空: 默认为 uname -n -n: 输出主机名 -a: 显示系统所有信息	uname uname -a	显示主机名称 显示系统所有的信息
ipcs 显示系统消息队列、共享内存和信号灯	ipcs -asmq ipcs [-s -m -q] -i id -a: 显示所有的信息 -s: 信号灯 -q: 消息队列 -m: 共享内存 空: 默认为显示 ipcs -a	ipcs ipcs -q ipcs -m ipcs -s	显示消息队列、信号灯、共享内存的信息 显示系统消息队列的信息 显示系统共享内存的信息 显示系统信号灯的信息
ipcrm 删除系统指定 id 号的消息队列、共享内存、信号灯	ipcrm [-q msqid] [-m shmid] [-s semid]	ipcrm -q 136 ipcrm -s 1970	删除 msqid 为 136 的消息队列 删除 semid 为 1970 的信号灯
alias 生成新命令或改变默认命令	alias 命令='命令内容' 定义放在用户目录下的 .profile 中	alias rm='rm -i' alias cdx='cd /home /user/src'	执行 rm aaa 相当于 rm -i aaa 执行 cdx 相当于 cd /home/user/src
time 显示命令执行时间	time 执行码	time ls	显示 ls 命令执行的时间
top 显示进程运行情况	空 描述显示进程运行 -n 秒数 间隔秒数 -u 用户 指定用户	top top -n 10 top -u root	默认秒数显示进程情况 每隔 10 秒显示进程情况 显示 root 用户进程运行情况
iostat 统计并输出 CPU 使用信息及特定设备或分区的 I/O	iostat -c 仅显示 CPU 使用情况统计信息 iostat -d 仅显示设备/分区使用情况统计信息 iostat -d 3 统计设备使用情况，并每隔 3 秒刷新一次 显示 I/O 的统计信息，显示信息各部分说明如下： tps 设备每秒收到的 I/O 传送请求数 Blk_read/s 设备每秒读入的块数量 Blk_wrtn/s 设备每秒写入的块数量 Blk_read 设备读入的总块数量 Blk_wrtn 设备写入的总块数量		

续表

命 令	参数及说明	例 子	附加说明
3.网络管理			
ftp 文件传输	ftp ip 地址 name >输入用户 passwd>输入密码 ftp>help 帮助命令 ftp>ascii 设置 ASCII 码方式传送 ftp>bin 设置二进制方式传送 ftp>cd 改变远程路径 ftp>lcd 改变本地路径 ftp>ls 查看远程路径内容 ftp>get file 从远程得到单个文件 ftp>mget xx* 从远程得到以 xx 开头的多个文件 ftp>put file1 向远程传输文件 file1 ftp>mput yy* 向远程传输以 yy 开头的多个文件 ftp>prompt on off 是否打开交互式问答 ftp>bye 退出 ftp		
telnet 远程操作	telnet ip 地址 login >输入用户 passwd >输入密码	telnet 127.0.0.1 login>user1 passwd>user1	利用 telnet 进入远程后，操作就像本地操作一样
ping 测试网络是否联通	ping ip 地址	ping 192.168.196.176	测试与该主机是否联通
netstat 检查整个 Linux 网络状态	a: 显示所有连接中的 socket -i: 显示接口信息表单 -l--listening: 显示监控中的服务器的 socket -n: 直接使用 IP 地址 -r: 显示 Routing Table -s: 显示网络工作信息统计表 -t: 显示 TCP 协议的连线状况	netstat -rn netstat -in netstat -s netstat -l netstat -t	查看路由表 查看接口信息 查看 tcp 统计信息 查看 listening 服务的套接口 查看 tcp 服务
traceroute 跟踪路由	traceroute ip 地址	traceroute 192.168.196.176	跟踪到该 ip 地址的路由
ifconfig 查看 ip 信息	空	ifconfig	查看 ip 地址及其信息 增删 ip 地址请使用图形界面
route 增加、修改、删除 路由信息	空 查看路由 route {add del flush} ip 地址 [掩码] [网关]	route route add default 192.168.1.1 route add -net 192. 56.76.0 netmask 255. 255.255.0 dev eth0	查看路由 增加默认网关为 192.168.1.1 在以太网 eth0 口上增加 192.56.76.0 为网关
ssh 安全外壳协议(SSH)	ssh 用户名@IP 地址 此命令相当于加密传送数据的 telnet	ssh root@192.16.1.1	以 ssh 登录该主机的 root 用户

续表

命 令	参数及说明	例 子	附加说明
scp 远程复制文件	scp [[user@]host1:] file1 路径 scp file2 [[user@]host2:] 路径	scp user@1.1.1.1:to /home/user2/tmp scp file1 user@1. 1.1.1:tmp	将远程用户主目录下 to 文件复制到当前主机/home/user2/tmp 下 将当前主机 file1 文件复制到远程主机 user 用户的 tmp 目录下
arp	空 查看 ip 的物理地址	arp	查看 IP 的物理地址信息
4. 压缩/解压			
说明：压缩/解压方式有多种，根据需要选择合适的一种即可。同类型压缩/解压需配对使用			
tar 对文件打包解包命令	-c 生成一个新的归档 -v 显示指令执行过程 -f 指令备份文件 -t 列出备份内容 -x 从备份文件还原文件 -z 使用 gzip 压缩，或使用 gunzip 解压 -j 使用 bzip2 格式压缩或解 压	tar cvf test.tar tmp file tar tvf test.tar tar xvf test.tar tar xvf test.tar file tar zcvf test.tar.gz tmp file tar zxvf test.tar.gz	将 tmp 目录和 file 文件打包到 test.tar 文件中 查看 test.tar 的打包内容 将 test.tar 文件解包 在 test.tar 文件中解压出单个 file 文件 将 tmp 目录和 file 文件打包并使用 gzip 压缩 对压缩的打包文件使用 gunzip 解压并解包
gzip	gzip 文件名 压缩文件 -a 以 ASCII 文字模式压缩 -d 解压 相当于 gunzip	gzip test.tar gzip -d test.tar	压缩 test.tar 文件 解压 test.tar 文件
gunzip	gunzip 文件名 解压文件 -a 以 ASCII 文件方式解压	gunzip test.tar.gz	对 test.tar.gz 文件解压
bzip2	bzip2 文件名 压缩文件 -d 解压 相当于 bunzip2	bzip2 test.tar bzip2 -d test.tar.bz2	使用 bzip2 压缩文件 对格式为 bz2 的压缩文件解压
bunzip2	bunzip2 文件名 解压文件	bunzip2 est.tar.bz2	对格式为 bz2 的压缩文件解压
compress	compress 文件名 压缩文件	compress test.tar	UNIX 环境中常见，较少使用
uncompress	uncompress 文件名 解压文件	uncompress test.tar.Z	UNIX 环境中常见，较少使用
5. 重定向命令			
xargs	命令 X xargs 命令 Y 将命令 X 的结果转换为命令 Y 的 参数输入	ls /etc/passwd xargs wc -l	比较加上和去掉 xargs 命令的两种输出结果

3. Linux 次常用命令

下面的命令没有常用命令使用的概率高，但在有些特定场合也经常用到，所以把它们归类为 Linux 次常用命令。Linux 次常用命令如表 1-4 所示。

表 1-4 Linux 次常用命令

命 令	参数及说明	例 子	附加说明
1. 文件及目录管理			
stat 显示 inode 内容	stat [文件或目录]	stat /tmp	显示/tmp 目录的 inode 内容
pg 分屏显示文件	分屏显示文件	pg file	分屏显示 file
cut 按列截取文件到标准输出	-c num1-num2 文件 显示字符 num1-num2 的列 -d 分界符 默认为 TAB -f num1-num2 显示 num1-num2 的列	cut -c20-30 ccc cut -f 1-3 -d': '/etc/passwd	显示 ccc 文件 20~30 字符的列 显示以 “: ” 为分隔符的 1~3 列
split 将一个大文件切分成若干个小文件	-l 按行数切分 -b 按字节数切分 -C 按字节数切分, 但切割时维护每行的完整性	man gcc > gcc.txt wc -l gcc.txt split 5000 gcc.txt	生成 gcc.txt 文件 统计 gcc.txt 的行数 每 5000 行切分成一个文件
paste 多个文件列合并	-d 指定列分割符, 默认为空格	paste fil1 fil2>fil3 paste -d "\t" file1 file2	把 fil1、fil2 合并到 fil3 中, 以 tab 键作为分隔符, 将 file1、file2 的列合并, 并将内容显示在屏幕上
ln 建立链接	ln 目标文件 链接文件名 ln -s 目标文件 链接文件名	ln aaa ln1 ln -s aaa ln2	建立一个 aaa 文件的硬链接 建立一个 aaa 文件的软链接
2. 系统管理			
trap 屏蔽中断信号	trap "" 中断信号列表	trap "" 1 2 3 20 trap "" HUP INT QUIT TSTP	屏蔽中断信号 1、2、3、20。 其中,1 代表 HUP,2 代表 INT, 3 代表 QUIT 与上面命令效果完全相同, 只不过利用宏代替数字
id 显示用户的用户 id 和组 id 号	空	id	显示用户的用户 id 和组 id 号
who am i 查看用户的终端号	空	who am i	查看用户终端号和 IP 地址
du 查看文件目录大小	du 目录 -a 显示所有的目录大小, 包括 0 块 -k 以 1KB 为单位显示大小 -h 以 MB、KB 为单位进行显示	du tmp du -akh tmp1	显示 tmp 目录下文件大小 以人们较为理解的格式显示文件大小
fsck 检查和修复 Linux 文件系统	空	fsck	修复 Linux 文件系统
sleep 睡眠秒数	sleep 数字	sleep 9	当前界面睡眠 9s

续表

命 令	参数及说明	例 子	附加说明
tr 字符替换	tr [选项] [设定 1] [设定 2] [设定 1 2]可使用正则表达式 -d 删除[设定 1]内容 -s 将设定 1 的重复内容替换成一个字符长度	tr -s h <test.txt tr -d ' '<test.txt tr 'a-z' 'A-Z' < test.txt	将 test.txt 中连续的 h 变成一个 h 删除 test.txt 中的空格 将 test.txt 中小写字母变成大写字母
tty 显示终端号	空	tty	显示终端号
stty 显示和设置终端参数	-a 显示所有的参数 stty 参数设置	stty -a stty -echo stty echo	显示终端所有的参数 禁止回显 打开回显
sudo 改变用户执行命令	sudo -u 用户 命令行	sudo -u root "date"	在当前用户下用 root 权限执行 date 命令
bc 计算操作	bc [数字][操作符][数字]	bc 5*6	进入 bc 工具 计算 5*6
skill 发送信号	skill [信号] [选项] -t 终端代号 -u 用户 -p 进程 pid 号 信号的三种写法：数字、全称、简称。如：9 SIGKILL KILL 都表示 9 号信号	skill -KILL -t pts/0 skill -STOP -u user1 user2	杀死终端pts/0下所有的进程 停止 user1、user2 用户所有的进程
fdisk 分区命令	fdisk -l 查看分区 fdisk 设备 分区操作	fdisk -l fdisk /dev/hda	查看分区 对/dev/hda 进行分区操作
mkfs.ext2 建立 ext2 文件系统	-b 文件块大小 -i 每隔多少空间建立一个 inode 节点	mkfs.ext2 -b 4096 -i 16384 /dev/ram5	在/dev/ram5 上建立 ext2 文件系统，块大写字为 4096B，每隔 16384B 建立一个 inode 节点
mount 将设备镜像到目录下	mount 设备 目录	mkdir mnt/cdrom mount dev/cdrom /mnt/cdrom	建立镜像目录 将光盘镜像到一目录下，镜像后即可像普通目录一样操作
umount 解除设备镜像	umount 设备 umount 镜像目录	umount /dev/cdrom umount /mnt/cdrom	解除光盘镜像 解除光盘镜像
finger 显示用户信息	finger 用户名	finger root	显示 root 用户信息
dd 复制文件并转换	if=输入文件名 of=输出文件 其他参数请用 dd --help 查看	dd if=/dev/hdb of=/dev/hdd	将本地的/dev/hdb 整盘备份到/dev/hdd
shutdown 关机	shutdown [选项] [时间] -k 分钟 告知几分钟后关机 -r 分钟 关机后重启 -h 分钟 关机后不重启	shutdown -k 3 shutdown -h 0 shutdown -r 1	告知所有的用户 3 分钟后将关机立即关机 1 分钟后关机并重启

续表

命 令	参数及说明	例 子	附加说明
reboot 关机并重启	空	reboot	关机并重启
sar	空 查看设备运行信息 -u 查看 CPU 和 I/O -q 查看 CPU -b 查看内存和 I/O -d 报告块设备	sar sar -u sar -q sar -b sar -d	同参数说明栏
vmstat	报告虚拟内存统计	vmstat	报告虚拟内存统计
nc 网络检测工具	-l 服务端侦听 -p 端口	服务端 nc -l -p 8000 客户端 nc 127.0.0.1 8000	一个作为服务端，一个作为客户端。分别在两个屏幕上输入字符，发现信息在两边传输
wget 从互联网上下载文件	空 下载该文件 -r 递归下载	wget www.baidu.com	下载百度主页
3. 用户与组操作（建议在界面上操作）			
useradd 建立用户 (root 权限)	useradd 用户名 -g 组名 -G 组名 -d Home 主目录 -p 密码 -m -s shell -g 主组 -G 次要组 -d 创建主目录 -p 密码 -u 强制设置一个 uid -m 生成主用户目录	groupadd ora useradd test -g ora -G root -d /home/oracle -p oracle -s /bin/bash -m	增加 ora 组 增加 test 用户，其主组为 ora，次要组为 root
userdel 删除用户 (root 权限)	userdel 用户名 删除用户 -r 删除用户的同时删除用户目录	userdel test userdel -r test	删除 test 用户 删除 test 用户的同时删除目录
groupadd 创建组 (root 权限)	groupadd 组名 增加组 -g 设置该组的 gid 号	groupadd ora -g 1000 groupadd oral	设置一个新组, gid 号为 1000 设置一个新组 oral
groupdel 删除组 (root 权限)	groupdel 组名 删除组	groupdel oral	删除组 oral

第 2 章

Linux常用实用工具

本章列出了 Linux 常用的几种实用工具，其中正则表达式符号的含义和 find 工具较为常用，对于 sed 的内容，读者只需了解即可，对 awk 则需重点掌握实例练习的打印记录部分。

2.1 正则表达式

正则表达式完成字符串的匹配搜索，可以对满足匹配条件的特定匹配串进行替换等处理。

在 UNIX 系统中，sed、grep、vi 等工具都使用 regexp（正则表达式英文缩写）规则。

开源软件的 perl、cygwin 也提供了在 Windows 系统中使用 regexp 规则的工具。

1. 正则表达式概述

空格、字母、数字可以直接作为正则表达式的字符匹配。对于特殊字符，可以使用\符号转义取得。

2. 正则表达式基本语法

表 2-1 列出了正则表达式的符号及其含义，这些符号是正则表达式最基本、最常用的语法。

表 2-1 正则表达式基本语法

符 号	含 义	举 例	匹 配
.	任何字符	a..	a 后面两个字符
?	修饰匹配次数为 0 次或 1 次	x?	0 个或 1 个 x
^	行首	^word	位于行首的 word
+	重复 1 次或多次	y+	1 或多个连续的 y
\$	行尾	x\$	以 x 结尾的行
		^HELLO\$	只包括 HELLO 的行
		^\$	不包括任何字符的行

续表

符 号	含 义	举 例	匹 配
*	重复 0 次或若干次	x* xx* .* w.*s	0 或若干连续的 x 1 或多个连续的 x 0 或若干个字符 以 w 开头、s 结尾的任何字符串
[字符表]	字符表中的任一字符	[tT] [a-z] [a-z A-Z]	小写字母 t 或大写字母 T 26 个任一小写字母 任一小写字母或大写字母
[! 字符表]	匹配不属于范围指定内的字符	[!a-z]	匹配不是小写字母的字符
[^ 字符表]	任一不在字符表中的字符	[^0-9] [^a-z A-Z]	任何非数字 任何非字母
\	转义字符，用来屏蔽一个元字符的特殊含义。因为有时在 Shell 中一些元字符有特殊含义。\\ 可以使其失去应有的意义	\\n \\n *	匹配换行符 匹配字符 n 匹配字符 *
{ min, max }	前导正规表达式最少重复 min 次，最多重复 max 次	x{1,5} [0-9]{3,9} [0-9]{3} [0-9]{3,}	最少 1 个，最多 5 个 x 3~9 个数字 正好 3 个数字 至少 3 个数字
(...)	将圆括号中匹配的字符串存储到下一个寄存器中 (1~9)	(more) and \1	匹配 more and more
表达式 1 表达式 2	匹配表达式 1 或表达式 2	hello happy	匹配 hello 或 happy

3. 正则表达支持 POSIX 字符集合

表 2-2 列出了正则表达式 POSIX 字符集合及其含义，POSIX 字符集合较少使用，读者只需了解即可。

表 2-2 正则表达式 POSIX 字符集合

POSIX 字符集合	说 明
[:alnum:]	任何一个字母或数字 (A ~ Z, a ~z, 0~9)
[:alpha:]	任何一个字母 (A~Z, a~z)
[:ascii:]	任何一个 ASCII 范围内的字符 (\\x00 ~ \\x7F)
[:cntrl:]	任何一个控制字符 (\\x00 ~ \\x1F, \\x7F)
[:digit:]	任何一个数字 (0 ~ 9)
[:print:]	任何一个可显示的 ASCII 字符 (\\x20 ~ \\x7E)
[:space:]	任何一个空白字符 (\\x09 ~ \\x0D, \\x20)
[:graph:]	任何一个可显示的 ASCII 字符，不包含空格 (\\x21 ~ \\x7E)
[:lower:]	任何一个小写字母 (a ~ z)
[:punct:]	可显示字符 [:print:] 中除去字母数字 [:alnum:]
[:upper:]	任何一个大写字母 (A ~ Z)
[:xdigit:]	任何一个十六进制数 (0 ~ 9, A ~ F, a ~ f)
[:blank:]	空格或者制表符 (\\x20, \\x09)

4. 正则表达式的转义字符

表 2-3 列出了正则表达式的转义字符及其含义，这些转义字符较少使用，读者只需了解即可。

表 2-3 正则表达式的转义字符

\onnn	如\o001，匹配一个八进制的 8bit 长的数值字符
\xnn	如\xFE，匹配一个十六进制的 8bit 长的数值字符
\unnnn	如\u00A9，匹配用 4 个十六进制数表示的一个 Unicode 字符
\t	ASCII 的制表符
\n	ASCII 的换行符
\r	ASCII 的回车符
\f	ASCII 的换页符
\w	匹配一个 word 字符串，即由字母、数字和下画线（_）组成的字符串
\W	与上相反，匹配不符合 word 规定的字符串
\s	匹配空格
\S	不是空格的其他字符
\d	匹配一个数字
\D	不是数字的其他字符
\b	一个单词的开始
\B	一个单词的结束

2.2 find 查找命令

find 是查找特定特征文件的一个工具。

find 常用命令为：

```
find /home/test -name aa.sh -print
```

表示在/home/test 目录下查找名称为 aa.sh 的文件，并把路径打印出来。

2.2.1 find 语法

1. find 命令语法形式

find 命令的语法形式如下：

```
find pathname -options [-print -exec -ok]
```

参数介绍如下。

- ① pathname: find 命令所查找的目录路径。可以用 . 表示当前目录，用 / 表示系统根目录。
- ② -options: 表示选项。
- ③ -print: find 命令将匹配的文件输出到标准输出。

④ **-exec**: `find` 命令对匹配的文件执行该参数所给出的 Shell 命令。相应命令的形式为 `'command'_{ } \;`。注意, `{ }`和`\;`之间有空格。

⑤ **-ok**: 它和**-exec**的作用相同,只不过以一种更安全的模式来执行该参数所给出的 Shell 命令,在执行每一个命令之前都会给出提示,让用户来确定是否执行。

2. find 命令选项 (options)

`find` 命令有很多选项或表达式,每一个选项前面跟随一个横杠 (-),下面是 `find` 的主要选项。

① **-name**: 按照文件名查找文件。

② **-perm**: 按照文件权限来查找文件。

③ **-prune**: 使用这一选项可以使 `find` 命令不在当前指定的目录中查找,如果同时使用了 **-depth** 选项,那么**-prune**选项将被 `find` 命令忽略。

④ **-user**: 按照文件属主来查找文件。

⑤ **-group**: 按照文件所属的组来查找文件。

⑥ **-mtime -n +n**: 按照文件的更改时间来查找文件, **-n** 表示文件更改时间距现在 `n` 天以内, **+n** 表示文件更改时间距现在 `n` 天以前。`find` 命令还有**-atime**和**-ctime**选项,但它们都和 **-mtime** 选项相似,所以在这里只介绍**-mtime**选项。

⑦ **-nogroup**: 查找无有效所属组的文件,即该文件所属的组在 `/etc/groups` 中不存在。

⑧ **-nouser**: 查找无有效属主的文件,即该文件的属主在 `/etc/passwd` 中不存在。

⑨ **-newer file1 !file2**: 查找更改时间比文件 `file1` 新但比文件 `file2` 旧的文件。

⑩ **-type**: 查找某一类型的文件,诸如:

■ **b**: 块设备文件。

■ **d**: 目录。

■ **c**: 字符设备文件。

■ **p**: 管道文件。

■ **l**: 符号链接文件。

■ **f**: 普通文件。

⑪ **-size n[c]**: 查找文件长度为 `n` 块的文件,带有 `c` 时表示文件长度以字节计算。

⑫ **-depth**: 在查找文件时,首先查找当前目录中的文件,然后在其子目录中查找。

⑬ `-fstype`: 查找位于某一类型文件系统上的文件, 这些文件系统类型通常可以在配置文件 `/etc/fstab` 中找到, 该配置文件中包含了本系统中有关文件系统的信息。

⑭ `-mount`: 在查找文件时不跨越文件系统 `mount` 点。

⑮ `-follow`: 如果 `find` 命令遇到符号链接文件, 就跟踪至链接所指向的文件。

⑯ `-cpio`: 对匹配的文件使用 `cpio` 命令, 将这些文件备份到磁带设备中。

2.2.2 find 实例练习

1. 使用 name 选项

① 查找 `$HOME` 目录下后缀以 `.txt` 结尾的文件。

```
$ find ~ -name "*.txt" -print
```

② 在当前目录及子目录中查找文件名以一个大写字母开头的文件。

```
$ find . -name "[A-Z]*" -print
```

③ 在 `/etc` 目录中查找文件名以 `host` 开头的文件。

```
$ find /etc -name "host*" -print
```

④ 查找文件名以两个小写字母开头, 后跟两个数字的文件, 如 `aa37.txt`。

```
$ find . -name "[a-z][a-z][0-9][0-9].txt" -print
```

2. 使用 perm 选项

在当前目录下查找文件权限位为 `755` 的文件。

```
$ find . -perm 755 -print
```

3. 忽略某个目录

希望在 `/apps` 目录下查找文件, 但不希望在 `/apps/bin` 目录下查找。

```
$ find /apps -name "/apps/bin" -prune -o -print
```

4. 使用 user 和 nouser 选项

① 在 `$HOME` 目录中查找文件属主为 `dave` 的文件。

```
$ find ~ -user dave -print
```

② 在 `/etc` 目录下查找文件属主为 `uucp` 的文件。

```
$ find /etc -user uucp -print
```

③ 为了查找属主账户已经被删除的文件, 可以使用 `-nouser` 选项。

```
$ find /home -nouser -print
```

5. 使用 nogroup 选项

要查找没有有效所属用户组的所有文件，可以使用 `nogroup` 选项，其格式如下：

```
$ find / -nogroup -print
```

6. 按照更改时间查找文件

① 在系统根目录下查找更改时间在 5 日以内的文件。

```
$ find / -mtime -5 -print
```

② 在 `/var/adm` 目录下查找更改时间在 3 日以前的文件。

```
$ find /var/adm -mtime +3 -print
```

7. 查找比某个文件新或旧的文件

查找某个时间点之前的文件，例如：

```
$ touch -t 09101119 dstamp —— 创建一个文件的时间为 9 月 10 日 11 点 19 分  
$ find . -newer dstamp -print —— 查找该时间点之前的文件
```

8. 使用 type 选项

① 查找 `/etc` 目录下所有的目录。

```
$ find /etc -type d -print
```

② 在 `/etc` 目录下查找所有的符号链接文件。

```
$ find /etc -type l -print
```

9. 使用 size 选项

① 在当前目录下查找文件长度大于 1MB 的文件。

```
$ find . -size +1000000c -print
```

② 在 `/home/apache` 目录下查找文件长度恰好为 100B 的文件。

```
$ find /home/apache -size 100c -print
```

③ 在当前目录下查找长度超过 10 块的文件。

```
$ find . -size +10 -print
```

10. 使用 exec 或 ok

当匹配到一些文件以后，可能希望对其进行某些操作，这时就可以使用 `-exec` 选项。`exec` 选项后面跟所要执行的命令，然后是一对花括号 `{}`、一个空格和一个反斜杠 `\`，最后是一个分号。

① 利用 `ls -l` 列出所找到的文件。

```
$ find . -type f -exec ls -l {} \;
```

② 删除 5 天以前的日志。

```
$ find . -name "*.LOG" -mtime +5 -ok rm {} \;  
$ find /home/hap/src/* -name "*.o" -exec rm {} \;
```

11. 使用 xargs

在使用 `find` 命令的 `-exec` 选项处理匹配到的文件时，`find` 命令将所有匹配到的文件一起传递给 `exec` 执行。但由于有些系统对能够传递给 `exec` 的命令长度有限制，这样在 `find` 命令运行几分钟之后，就会出现溢出错误。错误信息通常是“参数列太长”或“参数列溢出”。这就是 `xargs` 命令的用处所在，常与 `find` 命令一起使用。

用 `grep` 命令在所有的普通文件中搜索 `device` 单词。

```
$ find / -type f -print | xargs grep "device"
```

2.3 sed

`sed` 是一个非交互性文本流编辑器，使用 `sed` 可以从文件和字符串中抽取所需信息。

2.3.1 sed 语法

1. sed 的三种调用格式

`sed` 具有以下三种调用格式。

① 命令行方式：

```
sed [选项] sed 命令 输入文件
```

② `sed` 脚本文件：

```
含 sed 命令脚本文件 [选项] 输入文件
```

③ `sed` 调用脚本文件：

```
sed [选项] -f 不含 sed 命令的脚本文件 输入文件
```

2. sed 的选项

`sed` 选项说明如下。

① `-n`：不打印，`sed` 不写编辑行到标准输出。`sed` 默认为打印所有的行，即编辑的行和未编辑的行，可以用 `p` 命令打印需要显示的行。

② `-c`：下一命令是编辑命令。使用多项编辑时可加入此选项，如果只用到一条 `sed` 命令，此选项无用，但指定它也没有关系。

③ `-f`：调用 `sed` 脚本文件。此选项表示 `sed` 的一个脚本文件支持所有的 `sed` 命令。例如：
`sed -f myscript.sed input_file`，这里的 `myscript.sed` 为支持 `sed` 命令的文件。

3. sed 定位文本的方式

利用 sed 浏览文件时，默认从第一行开始，有以下两种形式定位文本。

- 使用行号：用一个数字表示该行号，使用两个数字表示行号范围。
- 使用正则表达式。

表 2-4 给出了 sed 定位文本的方式。

表 2-4 sed 定位文本的方式

参 数	说 明	例 子	例子说明
x	表示当前操作行号	7	表示操作第 7 行
x,y	操作 x~y 行	7,70	对第 7~70 行进行操作
/pattern/	查询 pattern 模式的行	/disk/ /[A-Z]/	查询 disk 的行 查询大写字母的行
x,y!	不包括 x~y 行	5,9!	不包括第 5~9 行

4. sed 基本编辑命令

表 2-5 列出了 sed 的编辑命令。

表 2-5 sed 编辑命令

命 令	说 明
p	打印匹配行
=	显示文件行号
d	删除定位行
a\ i\ c\ s r w q l { n g	在定位行号后附加新文本信息 在定位行号后插入新文本信息 用新文本替换定位文本 使用替换模式替换相应模式 从另一个文件中读文本 写文本到一个文件 第一个模式匹配完成后推出或立即推出 显示与八进制 ASCII 代码等价的控制字符 在定位行执行的命令组 从另一个文件中读文本下一行，并附加在下一行 全文

5. sed 的常用操作

表 2-6 列出了 sed 的一些常用操作，其中，[]表示空格，[]表示 tab 键。

表 2-6 sed 的一些常用操作

操 作	说 明
'/\.\$/d'	删除以句点结尾的行
'/abcd/d'	删除包含 abcd 的行

续表

操 作	说 明
's/[][]*[/][/g'	用一个空格替换两个和两个以上的空格
's/^[][]*[/g'	将行首空格替换成空，即删除行首空格
's/\.[][]*[/][/g'	删除句点后跟两个或更多的空格，并以一个空格替代
'/^\$/d'	删除空行
'1d'	删除第一行
'\$d'	删除最后一行
's/_*[/g'	删除横线-----
's/^[.]/g'	删除第一个字符
's/^[^/]*[/g'	从路径中删除第一个 /
's/[][]/[]/g'	删除所有的空格，并用 tab 键替代
's/^[]*/g'	删除行首所有的 tab 键

2.3.2 sed 实例练习

1. sed 练习文档

使用 vi sed.txt 命令录入如下内容，然后保存。

```
BUGS Setuid shell scripts should be avoided at all costs.
    PS1, PS2, and PS4 should be subject to parameter

BSD          January 19, 2003
```

2. 打印范围

(1) 打印第 2 行

```
$ Linux: sed -n '2p' sed.txt
    PS1, PS2, and PS4 should be subject to parameter
```

(2) 打印第 1~3 行

```
$ Linux: sed -n '1,3p' sed.txt
BUGS Setuid shell scripts should be avoided at all costs.
    PS1, PS2, and PS4 should be subject to parameter
```

(3) 将第 1~3 行输入到 sed.tmp 中

```
$ Linux:sed -n '1,3p' sed.txt > sed.tmp
```

(4) 显示全文

```
$ Linux: sed -n '1,$p' sed.txt
```

(5) 打印首行

```
$ Linux: sed -n '1p'sed.txt
```

(6) 打印最后一行

```
$ Linux: sed -n '$p' sed.txt
```

(7) -n 参数意义

测试 `sed '2p' sed.txt`。我们希望输出第 2 行，但 `sed` 默认输出编辑文本，所以输出了全部文本，并把第 2 行又多输出一次。所以，`sed` 一般要加上 `-n` 参数，让标准输入不输出到屏幕上，只输出产生的结果。

3. 使用模式查询

(1) 查找 should 的行

```
$ Linux: sed -n '/should/'p sed.txt
BUGS Setuid shell scripts should be avoided at all costs.
    PS1, PS2, and PS4 should be subject to parameter
```

(2) 输出匹配的行号

```
$ Linux: sed -n '/should/= ' sed.txt
1
2
```

4. 特殊字符查询

说明：特殊字符匹配需要用 `\` 进行转义。

若要查找有 `.` 的行，则命令如下：

```
$ Linux: sed -n '/\./'p sed.txt
BUGS Setuid shell scripts should be avoided at all costs.
```

5. 删除文本

(1) 删除第 1~3 行

```
$ Linux: sed '1,3d' sed.txt
BSD          January 19, 2003
```

(2) 删除含有 should 的行

```
$ Linux: sed '/should/d' sed.txt

BSD          January 19, 2003
```

6. 替换文本

(1) 替换每行第一个单词

```
$ Linux:sed 's/PS/ps/' sed.txt
BUGS Setuid shell scripts should be avoided at all costs.
    ps1, PS2, and PS4 should be subject to parameter

BSD          January 19, 2003
```

(2) 全文替换 (/g 参数)

```
$ Linux:sed 's/PS/ps/g' sed.txt
BUGS Setuid shell scripts should be avoided at all costs.
    ps1, ps2, and ps4 should be subject to parameter

BSD          January 19, 2003
```

(3) 指定行替换

```
$ Linux:sed '1,3 s/should/SHOULD/g' sed.txt
BUGS Setuid shell scripts SHOULD be avoided at all costs.
    ps1, ps2, and ps4 SHOULD be subject to parameter

BSD          January 19, 2003
```

(4) 指定行范围替换

```
$ Linux:sed '1 s/should/SHOULD/g' sed.txt
BUGS Setuid shell scripts SHOULD be avoided at all costs.
    ps1, ps2, and ps4 should be subject to parameter

BSD          January 19, 2003
```

7. sed 脚本文件

(1) 含 sed 命令的脚本 (sed 执行的第二种方式)

Shell 中 #! 表示脚本使用哪种命令执行。

```
$ Linux:cat appsed.sed
#!/bin/sed -f
/should/ a\
Then suddenly it happened.
$ Linux:chmod +x appsed.sed
$ Linux:./appsed.sed sed.txt
BUGS Setuid shell scripts should be avoided at all costs.
Then suddenly it happened.
    PS1, PS2, and PS4 should be subject to parameter
Then suddenly it happened.

BSD          January 19, 2003
```

(2) 不含 sed 命令脚本 (sed 执行的第三种方式)

```
$ Linux:cat app.sed
3 a\
Then suddenly it happened.
$ Linux:chmod +x app.sed
$ Linux:sed -f app.sed sed.txt
BUGS Setuid shell scripts should be avoided at all costs.
    PS1, PS2, and PS4 should be subject to parameter

Then suddenly it happened.
BSD          January 19, 2003
```

8. 控制字符的输入

在使用 `sed` 命令时，有时需要对控制字符（回车、`Esc` 等）进行操作，其输入方法如下，以输入回车（`^M`）为例。

- ① 同时按下 `Ctrl` 键和 `V` 键。
- ② 按下 `M` 键，释放 `Ctrl` 键。
- ③ 回车字符输入完成。
- ④ 控制字符替换与其他普通字符相同。

2.4 awk

`awk` 是一种编程语言，对文本和数据进行处理，支持正则表达式，突出特点是对文本列的操作。

`awk` 有三个不同版本：`awk`、`nawk` 和 `gawk`，三个版本功能基本相同，`gawk` 是 `awk` 的 GNU 版本，未做特别说明，一般指 `gawk`。

`awk` 语言的最基本功能是在文件或字符串中基于指定规则来分解抽取信息，也可以基于指定的规则来输出数据。

2.4.1 awk 语法

1. awk 调用的三种方式

`awk` 调用有三种方式，分别为命令行方式、脚本执行方式和命令行调用脚本执行方式，三种方式具体说明如下。

(1) `awk [option] 'awk_script' input_file1 [input_file2 ...]`

`awk` 的常用选项 `option` 有：

- ① `-F fs`：使用 `fs` 作为输入记录的字段分隔符，如果省略该选项，`awk` 使用环境变量 `FS` 的值（默认为空格）。
- ② `-f filename`：从文件 `filename` 中读取 `awk_script`（`awk` 脚本）。
- ③ `-v var=value`：为 `awk_script` 设置变量。

(2) 将 `awk_script` 放入脚本文件并以 `#!/bin/awk -f` 作为首行，给予该脚本可执行权限，然后在 `Shell` 命令行下通过键入该脚本的脚本名进行执行。

(3) 将所有的 `awk_script` 写入到一个单独脚本文件，然后调用“`awk -f awk 脚本文件 输入文件列表`”进行执行。

2. awk 样例文本

为了在语法中举例，先建立两个 `awk` 文本，名称分别为 `grade.txt` 和 `passwd`。

grade.txt 文本文件记录了一个柔道信息库，此文本文件有七个域（即七列），域意义分别为（1）名字、（2）升段日期、（3）学生序号、（4）腰带级别、（5）年龄、（6）目前比赛积分、（7）比赛最高分。

grade.txt 内容如下：

```
M.Tansley 05/99 48311 Green      8  40 44
J.Lulu    06/99 48317 green      9  24 26
P.Bunny   02/99 48     Yellow     12  35 28
J.Troll   07/99 4842  Brown-3    12  26 26
L.Tansley 05/99 4712  Brown-2    12  30 28
```

passwd 以 “:” 分隔每列，文件内容如下：

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
```

3. 记录和域

（1）域分隔符

域分隔符：文本列与列的分隔符，默认是空格或 Tab。内置变量 FS 保存输入域分隔符值。
输出域的分隔符默认是一个空格，保存在内置变量 OFS 中。

修改输入域分隔符方法为：awk -F '[:\t]' '{print \$1,\$3}' passwd，此例表示以空格、冒号和 Tab 键为分隔符。

（2）记录

记录：awk 把每一个以换行符结束的行称为一个记录。
记录分隔符：默认的输入和输出的记录分隔符都是换行，保存在内置变量 ORS 和 RS 中。
变量 NR：一个计数器，每处理完一条记录，NR 的值就增加 1。

例：awk '{print NR,\$0}' grade.txt，这样例将打印行号和整行内容。

（3）域

域：记录中每一列叫做域。

域标志顺序为\$1,\$2,...,\$n。\$0 表示整条记录，\$1 表示当前行第 1 列，\$n 表示当前行第 n 列。

域之间的分隔符默认为空格或 Tab，可以用-F 参数修改域分隔符。

例：awk '{print \$1,\$3}' grade.txt

此样例为打印此文本第 1 列和第 3 列内容。\$1 与\$3 之间的逗号将使两列产生空格分隔符，若无逗号则两列会连在一起。

例: `awk '{ print $1 "|" $3}' grade.txt > grade1.txt`

此样例让两列用|分隔, 并输出到 grade1.txt 文件中。

4. awk_script 说明

(1) 模式与操作语法说明

awk 脚本 (awk_script) 由模式和操作组成。

格式为: `pattern {action}`, 其中, `pattern` 为模式, `action` 为操作。

两者是可选的, 如果没有 `pattern`, 则 `action` 应用到全部记录, 如果没有 `action`, 则输出匹配的全部记录。

默认情况下, 每一个输入行都是一条记录, 但用户可通过 `RS` 变量指定不同的分隔符进行分隔。

(2) 模式种类

① /正则表达式/: 使用正则表达式通配符的扩展集。常用通配符如下:

`\ ^ $. [] | () * //` 通用的 `regexp` 元字符。

`+` : 匹配其前的单个字符一次以上。

`?` : 匹配其前的单个字符 1 次或 0 次。

② 关系表达式。

③ 模式匹配表达式: 用运算符 `~` (匹配) 和 `!~` (不匹配)。

④ 范围模式: 指定一个行的范围, 该语法不能包括 `BEGIN` 和 `END` 模式。

⑤ `BEGIN`: 让用户指定在第一条输入记录被处理之前所发生的动作, 通常可在这里设置全局变量。

⑥ `END`: 让用户在最后一行输入记录被读取之后发生的动作。

(3) 操作种类

操作由一个或多个命令、函数、表达式组成, 之间由换行符或分号隔开, 并位于大括号内, 主要有四部分: 变量或数组赋值、输出命令、内置函数、控制流命令。

5. awk 的运行过程

(1) awk_script 的组成

awk_script 可以由一条或多条 awk_cmd 组成, 两条 awk_cmd 之间一般以换行分隔。

awk_cmd 由两部分组成: `awk_pattern { action }`。

awk_script 可以分成多行书写, 必须确保整个 awk_script 被单引号引起来。

(2) awk 命令的一般形式

awk 命令的一般形式如下：

```
awk ' BEGIN { action }
awk_pattern1 { action }
.....
awk_patternN { action }
END { action }
' inputfile
```

其中，BEGIN { action } 和 END { action } 是可选的。

(3) awk 的运行过程

- ① 如果 BEGIN 区块存在，awk 执行它指定的 action。
- ② awk 从输入文件中读取一行，称为一条输入记录(如果输入文件省略，将从标准输入读取)。
- ③ awk 将读入的记录分割成字段，将第 1 个字段放入变量\$1 中，第 2 个字段放入\$2，依此类推，\$0 表示整条记录。字段分隔符使用变量 IFS 或由参数指定。
- ④ 把当前输入记录依次与每一个 awk_cmd 中 awk_pattern 比较，看是否匹配，如果相匹配，就执行对应的 action，如果不匹配，就跳过对应的 action，直到比较完所有的 awk_cmd。
- ⑤ 当一条输入记录比较了所有的 awk_cmd 后，awk 读取输入的下一行，继续重复步骤③和④，这个过程一直持续，直到 awk 读取到文件尾。
- ⑥ 当 awk 读完所有的输入行后，如果存在 END，就执行相应的 action。

(4) inputfile 可以是多于一个文件的文件列表，awk 将按顺序处理列表中的每个文件。

6. awk 内置变量与内置函数说明

(1) awk 的内置变量

表 2-7 列出了 awk 内置变量，这些内置变量表示特定的含义，可以在 awk 脚本中直接使用。

表 2-7 awk 内置变量表

变 量	描 述
\$n	当前记录的第 n 个字段，字段间由 FS 分隔
\$0	完整的一条输入记录
ARGC	命令行参数的数目
ARGIND	命令行中当前文件的位置（从 0 开始算）
ARGV	包含命令行参数的数组
CONVFMT	数字转换格式（默认值为%.6g）
ENVIRON	环境变量关联数组
ERRNO	最后一个系统错误的描述
FIELDWIDTHS	字段宽度列表（用空格键分隔）

续表

变 量	描 述
FILENAME	当前文件名
FNR	同 NR，但相对于当前文件
FS	字段分隔符（默认是空格）
IGNORECASE	如果为真，则进行忽略大小写的匹配
NF	当前记录中的字段数
NR	当前记录数
OFMT	数字的输出格式（默认值是%.6g）
OFS	输出字段分隔符（默认值是一个空格）
ORS	输出记录分隔符（默认值是一个换行符）
RLENGTH	由 match 函数所匹配的字符串的长度
RS	记录分隔符（默认是一个换行符）
RSTART	由 match 函数所匹配的字符串的第一个位置
SUBSEP	数组下标分隔符（默认值是\034）

（2）awk 内置字符串函数

表 2-8 列出了 awk 内置字符串函数及其说明。

表 2-8 awk 内置字符串函数表

函 数	函数说明
gsub(r,s)	在整个\$0 中用 s 替代 r
gsub(r,s,t)	在整个 t 中用 s 替代 r
index(s,t)	返回 s 中字符串 t 的第一位置
length(s)	返回 s 长度
match(s,r)	测试 s 是否包含匹配 r 的字符串
split(s,a,fs)	用 fs 作为分隔符将 s 分成序列 a
sprint(fmt,exp)	返回经 fmt 格式化后的 exp
sub(r,s)	用\$0 中最左边最长的匹配 r 的子串代替 s
substr(s,p)	返回字符串 s 中从 p 开始的后缀部分
substr(s,p,n)	返回字符串 s 中从 p 开始，长度为 n 的后缀部分

7. awk 编程语法详细说明

（1）awk 变量

① 在 awk 中，变量不需要定义就可以直接使用，变量类型可以是数字或字符串。

② awk 赋值格式为：Variable=expression。例如，\$awk '\$1~/test/{count=\$2+\$3; print count}' test，上式的作用是，awk 先扫描第一个域，一旦 test 匹配，就把第二个域的值加上第三个域的值，并把结果赋值给变量 count，最后打印出来。

③ 域变量可被赋值和修改。例如，\$awk '{ \$2=100+\$1; print \$2 }' test，上式表示如果第二个域不存在，awk 将计算表达式 100 加\$1 的值，并将其赋值给\$2，如果第二个域存在，

则用表达式的值覆盖\$2 原来的值。

④ 可以使用内置变量。例如, `$awk '{ if($1=="MARY"){print NR,$1,$2,$NF}}'` test, 该例含义为打印 MARY 的记录数、第一个域、第二个域和最后一个域。

(2) awk 数组

awk 中数组叫做关联数组, 因为下标可以是数也可以是字符串。awk 中的数组不必提前声明, 也不必声明大小。数组元素用 0 或空串来初始化, 这根据上下文而定。

① 可以用数值作为数组索引(下标)。

```
Myarray[1]= "xu sihua"  
Myarray[2]= "780927"
```

② 可以用字符串作为数组索引(下标)。

```
Myarray["first"]="xu"  
Myarray["birth"]="780927"
```

(3) awk 语句

awk 有如下几种语句:

① 条件语句: if 语句、if-else 语句, 用法与 C 语言同名语句相同。

② 循环语句: while 循环、for 循环、do-while 循环、special for 循环。while 循环、do-while 循环、for 循环语句用法与 C 语言中同名语句相同。

③ 跳转语句: break 和 continue 语句, 作用和语法也与 C 语言同名语句相同。

④ next 语句: 从输入文件中读取一行, 然后从头开始执行 awk 脚本。

⑤ exit 语句: 用于结束 awk 程序, 但不会略过 END 块。退出状态为 0 代表成功, 非零值表示出错。

⑥ special for 循环语法: 与 Shell 语法 for 循环类似, 表示变量在数组中开始循环。special for 语法格式如下:

```
for (item in arrayname){  
    print arrayname[item]  
}
```

(4) awk 函数

awk 函数分为系统自带函数和用户自定义函数。print 函数和 printf 函数为 awk 的系统自带打印函数。其中, print 函数为 awk 常见的打印函数, 如果需要打印复杂的格式, 可以使用 printf 函数, 此函数用法几乎与 C 语言中 printf 函数相同。

在 awk 中自定义函数格式如下:

```
function name ( parameter, parameter, parameter, ... )
{
    statements
    return expression
}
```

(5) awk 特殊字符

表 2-9 列出了 awk 特殊字符及其说明。

表 2-9 awk 特殊字符表

特殊字符	说 明
\b	退格键
\f	走纸换页
\n	新行
\r	回车键
\t	Tab 键
\ddd	八进制
\c	任意其他特殊字符，如\\为反斜线符号

(6) awk 运算符

表 2-10 列出了 awk 运算符及其说明。

表 2-10 awk 运算符表

运 算 符	描 述
= += -= *= /= %= ^= **=	赋值
?:	条件表达式
	逻辑或
&&	逻辑与
~	匹配正则表达式
~!	不匹配正则表达式
< <= > >= != ==	关系运算符
空格	连接
+ -	加、减
* / &	乘、除与求余
+ - !	一元加、减和逻辑非
^ ***	求幂
++ --	增加或减少，作为前缀或后缀
\$	字段引用
in	数组成员

(7) awk 注意事项

- ① 确保整个 awk_script 用单引号引起来。
- ② 确保 awk_script 内所有引号成对出现。

- ③ 确保用花括号引起动作语句，用圆括号括起条件语句。
- ④ 如果使用字符串，一定要保证字符串被双引号引起来（在模式中除外）。

2.4.2 awk 实例练习

1. awk 的常见操作

下面是实际工作中利用 `awk` 工具常用的两种操作。

- ① 默认分隔符为空格打印相应列：`awk '{print $1,$3}' grade.txt`
- ② 指定分隔符打印相应列：`awk -F '[:\t]' '{print $1,$3}' passwd`

2. 打印记录

(1) 打印 `grade.txt` 所有记录

```
$ Linux: awk '{print $0 }' grade.txt
M.Tansley 05/99 48311 Green      8  40 44
J.Lulu    06/99 48317 green      9  24 26
P.Bunny   02/99 48     Yellow    12 35 28
J.Troll   07/99 4842  Brown-3   12 26 26
L.Tansley 05/99 4712  Brown-2   12 30 28
```

(2) 打印 `passwd` 的第 1 列和第 3 列

```
$ Linux: awk -F: '{print $1, $3}' passwd
root 0
daemon 1
```

(3) 打印报告头

```
$ Linux:awk 'BEGIN {print "Name Belt\n-----"}
{print $1"\t"$3}' grade.txt
```

结果如下：

```
Name Belt
-----
M.Tansley      48311
J.Lulu 48317
P.Bunny 48
J.Troll 4842
L.Tansley      4712
```

(4) 打印报告尾

```
$ Linux:awk 'BEGIN { print "Name \n ----"}{print $1} END { print "end-of-
report" }' grade.txt
Name
-----
M.Tansley
J.Lulu
P.Bunny
```

```
J.Troll
L.Tansley
end-of-report
```

3. 条件运算符

(1) 匹配

匹配 grade.txt 文本域 4 为 Brown 的项。

```
$ Linux:awk '{if ( $4~/Brown/) print $0}' grade.txt
J.Troll    07/99  4842   Brown-3   12  26  26
L.Tansley  05/99  4712   Brown-2   12  30  28
```

不匹配使用 “!~”。

(2) 精确匹配

精确匹配第 3 个域为 48。

```
$ Linux: awk '$3=="48" {print $0}' grade.txt
P.Bunny    02/99  48     Yellow    12  35  28
```

精确不匹配使用 “!=”。

(3) 字段比较

> (大于)、< (小于)、<= (小于等于)、>= (大于等于) 使用方法差不多，故在这里只举一例：如果第 7 个域大于第 6 个域，则打印第 1 个域和提示信息。

```
$ Linux: awk '{if ($6 < $7){ print $1, "Try better at the next comp" }}' grade.txt
M.Tansley Try better at the next comp
J.Lulu Try better at the next comp
```

4. 利用正则表达式

(1) 匹配字符大小写

匹配 Green 或者 green。

```
$ Linux: awk '/[Gg]reen/' grade.txt
M.Tansley  05/99  48311  Green     8  40  44
J.Lulu     06/99  48317  green     9  24  26
```

(2) 匹配任意字符

匹配第一个域的第 4 个字符为 a，前三个字符为任意。本例^符号代表行首，.表示匹配任意字符。

```
$ Linux: awk '$1~/^...a/' grade.txt
M.Tansley  05/99  48311  Green     8  40  44
L.Tansley  05/99  4712   Brown-2   12  30  28
```

(3) 或关系匹配

匹配 Yellow 或 Brown 的行。

```
$ Linux: awk '$0~/ (Yellow|Brown)/' grade.txt
P.Bunny    02/99  48      Yellow    12  35  28
J.Troll    07/99 4842    Brown-3   12  26  26
L.Tansley  05/99 4712    Brown-2   12  30  28
```

(4) 复合匹配关系

复合匹配关系有如下三种：

① && (AND)：语句两边必须同时匹配为真。

② || (OR)：语句至少一边匹配为真。

③ ! (非)：进行逆运算。

打印出第 1 域为 “P.Bunny” 和第 4 域为 “Yellow” 的行。

```
$ Linux: awk '{ if ( $1=="P.Bunny" && $4=="Yellow" ) print $0 }' grade.txt
P.Bunny    02/99  48      Yellow    12  35  28
```

5. 使用内置变量

下面用到的内置变量 NF 为列数，内置变量 NR 为行号。

```
$ Linux: awk '{print NF,NR,$0} END {print FILENAME}' grade.txt
7 1 M.Tansley 05/99 48311 Green      8  40  44
7 2 J.Lulu    06/99 48317 green     9  24  26
7 3 P.Bunny   02/99  48      Yellow   12  35  28
7 4 J.Troll   07/99 4842    Brown-3  12  26  26
7 5 L.Tansley 05/99 4712    Brown-2  12  30  28
grade.txt
```

6. 使用内置字符串函数

(1) gsub 函数

把有 4842 的行中的 4842 替换为 4899。

```
$ Linux: awk 'gsub (/4842/,4899) {print $0}' grade.txt
J.Troll    07/99 4899    Brown-3   12  26  26
```

(2) index 函数

找出字符串的位置。

```
$ Linux: awk 'BEGIN {print index("Bunny","ny")}' grade.txt
4
```

(3) match 函数

```
$ Linux:awk 'BEGIN {print match("ABCD",/C/)}' grade.txt
3
```

(4) split 函数

```
$ Linux: awk 'BEGIN {print split("123#456#678",myarray, "#")}'  
3
```

这个例子中，split 返回数组 myarray 的下标数。myarray 取值如下：myarray[1]为 123、myarray[2]为 456、myarray[3]为 678。

7. 输出函数 printf

利用 printf 函数进行复杂格式输出。

```
$ Linux: awk '{ printf "%-15s %s\n", $1,$3}' grade.txt  
M.Tansley      48311  
J.Lulu         48317  
P.Bunny        48  
J.Troll        4842  
L.Tansley      4712
```

第 3 章

Shell编程

Shell 编程在 Linux 从业中经常要用到,而且是在 Linux 行业网上招聘中经常要求的技能,读者需要重点掌握。

Shell 的中文意思是“外壳”,通俗地讲,Shell 是一个交互编程接口,也是一个命令解释语言,还是一种命令语言解释器,可以用 Shell 编写各种脚本工具。解释型语言的特点是对源文件进行边识别翻译边执行,不会直接生成二进制机器码,Shell 就是解释型语言。Shell 脚本文件交给 Shell 解释器进行解析,Shell 解释器把 Shell 脚本文件解析成一条条语句,然后调用每条语句相应的系统命令进行执行。Shell 作为一种命令语言解释器,内置了大量的命令集,涵盖了当前 Linux 中的所有命令。Shell 脚本的执行流程是 Shell 解释器顺序读取每一行命令,识别成一条条的 Linux 系统指令,然后调用 Linux 相应的命令接口生成执行结果。

Shell 除了作为命令解释程序以外,还是一种高级程序设计语言,它有变量、关键字,有各种控制语句、支持函数模块,有自己的语法结构,利用 Shell 程序设计语言可以编写出功能很强但代码简单的程序。

Shell 有 Bourne (简称 B) Shell、Korn Shell、C Shell 三种类型,三种 Shell 的功能大同小异,用得最多的还是 B Shell。Shell 脚本文件头可用 `#!/bin/sh` 说明脚本用哪一种 Shell 执行, `#!` 表示使用哪一种解释器执行当前文本, `/bin/sh` 是指 B Shell 解释器。若文件头无 `#!` 说明用哪种解释器执行文本,系统会选择 Shell 环境变量作为此文本的解释器。Shell 的注释以 `#` 号开头,后面接注释文字。

3.1 Shell 环境变量

3.1.1 环境变量说明

环境变量用于描述该用户的操作环境下特定意义的变量,或者说是通过设置环境变量来配置用户的操作环境。Linux 中的环境变量包括:用户所使用的 Shell 类型、工作的主目录、登录方式等。

环境变量分为系统环境变量和用户自定义环境变量。系统环境变量为在系统中有特定意义的环境变量，而且在不定义时也存在，系统环境变量可以重新进行赋值，如 PATH 环境变量就是系统环境变量，表明用户搜索执行码时所用到的路径。

Shell 用户环境变量是每一个 Linux 用户定义在 .profile 或 .bash_profile 中生效的变量，同时还包括 .bash_profile 中包含执行脚本的环境变量。

环境变量包括定义和导出生效两部分，定义 INFORMIXDIR 环境变量，如 INFORMIXDIR=/usr/informix，导出生效，如 export INFORMIXDIR。只有导出生效的环境变量才能被引用，引用时需要在变量前面加\$符号。

环境变量定义和导出有如下两种格式：

- ① name=value; export name
- ② name=value
export name

unset 命令用来删除环境变量，如 unset USERNAME 是删除 USERNAME 变量。

环境变量使用 alias 进行变量重定义，重定义的环境变量不需要用 export 导出，定义方法如 alias ygp='cd ~/public/ygp'。

3.1.2 用户常用的系统环境变量

表 3-1 列出了用户常用的系统环境变量，这些变量可以在 .bash_profile 中重新赋值，以便适应用户环境客户化的需要。

表 3-1 用户常用系统环境变量表

变 量 名	意 义
PWD	当前用户的工作目录
HOME	用户主目录的路径全名
LOG NAME	用户的登录名
SHELL	当前所使用的 Shell
PATH	搜索执行码的路径
PS1	命令行提示符
LANG	定义语言编码方式
EXINIT	保存 vi 编辑初始化设置选项，如设置行号，设置 tab 为 4 个空格，设置方法如下： EXINIT='set nu tab=4';export EXINIT

下面是 env.sh 脚本，用来输出环境变量。

```
#!/bin/sh
echo "PWD: "$PWD
echo "path: "$PATH
echo "Logname: "$LOGNAME
echo "Shell: "$SHELL
echo "HOME: "$HOME
```


\$ chmod u+x env.sh ——增加执行权限

\$./env.sh ——执行，执行结果如下：

```
PWD:/home/zfb
path:/home/zfb/extra/bin:/home/zfb/extra/bin:/usr/lib/jdk/bin:/usr/kerberos/
bin
Logname:zfb
Shell:/bin/bash
HOME:/home/zfb
```

3.1.3 用户登录脚本示例

1. 命令行提示符

下文约定脚本行首\$为命令行的提示符。

```
$pwd
/home/zfb/public/ygp/shell
```

2. 用户.bash_profile 脚本部分内容

下面是一个用户下的.bash_profile 脚本，后文会对此脚本进行解释说明。

```
INFORMIXDIR=/usr/informix
PATH=$PATH:$INFORMIXDIR/bin:$HOME/bin:.
PATH=$PATH:/opt/subversion/bin.
PS1='Linux 开发': '$PWD>'
# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi
alias rm='rm -i'
alias ygp='cd /home/zfb/public/ygp'
export PATH INFORMIXDIR
```

(1) 脚本具体说明如下。

上面的示例中 PATH、INFORMIXDIR 都是环境变量，其中，PATH 是系统环境变量，INFORMIXDIR 是用户自定义环境变量，环境变量必须用 export 命令导出才能生效。

环境变量可以递归赋值，上面示例中的 PATH 就进行了递归赋值。

上面示例中 HOME 为系统环境变量，其路径值是由增加用户时进行指定的。

PATH 指出此用户下执行程序的搜索路径，不同路径名用“:”分开，以“.”结束搜索路径。当我们敲入一执行码时，./test 系统会根据 PATH 路径去寻找该执行码。

.bash_profile 可以包含其他可执行脚本，如上面.bashrc 文件在当前目录存在，则解释执行。

alias 是对变量进行重定义的命令。

(2) `.bash_profile` 文件何时执行？

① 用户登录时立即执行。

② 可以手工执行，在主目录下用 `./bash_profile` 来进行重新执行操作。

3.2 Shell 的符号、变量及运行

3.2.1 Shell 中的符号及其含义

在 Shell 中，内置了许多特定符号，这些符号分别代表着特定的含义，下面是对这些符号的说明。

- `*`: 匹配 0 个和多个字符组成的串。
- `?`: 匹配单个字符。
- `[]`: 匹配的字符范围或列表。例如，`$ls [a-c]*`，将列出以 `a~c` 范围内字符开头的的所有文件，`$ls [e,m,t]*` 将列出以 `e、m 或 t` 开头的的所有文件。
- `>`: 为重定向覆盖输出。
- `<`: 为重定向输入。
- `>>`: 为重定向添加。
- `|`: 管道命令，左边的输出作为右边的输入，如 `ls *.c|wc -l`。
- `$#`: 传送给命令 Shell 的参数序号。
- `$-`: 在 Shell 启动或使用 `set` 命令时提供选项。
- `$?`: 上一条命令执行后返回的值。
- `$$`: 当前 Shell 的进程号。
- `$!`: 上一个子进程的进程号。
- `$@`: 所有的参数，每个都用双引号引起，以 (`"$1"$2"..."`) 的形式保存所有输入的命令行参数。
- `$*`: 所有的参数，用一个双引号引起整体，以 (`"$1 $2..."`) 的形式保存所有输入的命令行参数。
- `$n`: 位置参数值，`n` 表示位置。
- `$0`: 当前 Shell 名。

- `$`: 引用某个变量。
- `#`: 注释符号。
- `&`: 后台命令。
- `&&` (布尔与) 与条件符号: 仅当其左边命令执行成功后, 才执行其右边命令。
- `||` (布尔或) 或条件符号: 仅当其左边命令执行不成功时, 才执行其右边命令。
- `!` (布尔非): 反转命令的退出状态值。
- `;`: 命令分隔符, 在一个命令行中依次执行各个命令。
- `"...":` 双引号表示除 `\`、`$`、`'` 和 `"` 外, 由双引号引起来的字符为普通字符。
- `'...':` 单引号引起来的字符均作为普通字符。
- ``...``: 命令替换, 倒引号引起来的字符串作为 Shell 命令执行。
- `~`: 表示主目录。
- `.` (内置句点): 执行命令。
- `..`: 表示上级目录。
- `[]` (内置表达式): 计算算术表达式的值, 相当于 `test`。
- `{ }`: 用来封装函数体。
- `\`: 表示转义字符。
- `<<`: 重定向输入。
- `<<-`: 重定向输入, 输入去掉行首的 Tab 键。

3.2.2 “反引号命令替换

“内部整体作为 Linux 命令执行输出, 例 3-1 和例 3-2 说明了反引号的用法。

【例 3-1】 命令行的使用

```
$wc -l `ls test.c`  
7 test.c
```

【例 3-2】 base.sh 脚本用例

(1) 编写测试脚本 `base.sh`

```
#!/bin/sh
```

```
#使用 basename 命令得到文件名
```

```
file=`basename $0`  
echo "file name : $file"  
  
#使用 pwd 命令得到当前路径  
path=`pwd`  
  
#将两个字符串连接起来  
path=$path/$file  
echo "full path : $path"  
exit 0
```

(2) 增加脚本执行权限

```
$chmod u+x base.sh
```

(3) 使用 ./base.sh, 执行结果如下:

```
file name : base.sh  
full path : /home/zfb/public/ygp/shell/base.sh
```

3.2.3 Shell 变量

1. 变量特点

Shell 中变量特点如下:

- ① 无需定义, 可直接使用。
- ② Shell 大小写敏感。
- ③ \$ 为 Shell 保留字符, 变量被其他变量引用时前面需要加 \$。
- ④ 变量赋值 “=” 两边是没有空格的, 不然会带来错误。

⑤ 如果在赋值语句中, 右边没有任何信息, 那么这个变量为一个空字符串; 另外仅定义声明而没有赋值的变量, 默认也是一个空字符串。

⑥ Shell 只有两种类型, 一种是整型数字, 一种字符串。整型数字必须所有位都为数字, 类型 Shell 解释器自动识别。

⑦ 如果一个变量中含空格、制表位、换行符, 则要用双引号引起, 不然会出错。

⑧ 字符串左右应加双引号 “”。

⑨ Shell 内置 9 个位置变量, \$1~\$9。

2. 引用变量三种方法

Shell 中引用变量有如下三种方法:

- ① 使用双引号引用变量

```
"$var"
```

② 使用大括号引用变量

```
{ $var }
```

③ 直接引用变量

```
$var
```

3. 用户变量赋值

(1) 用户变量赋值种类

用户变量赋值有如下 4 种方法：

① 直接赋值

```
user=meng      #字符串赋值
null=          #空串赋值
number=12345   #数字赋值
```

② 变量赋值

```
var1=$user
var2=$var1
```

③ read 读入

```
read 变量1 [变量2]
read var1 var2
#当输入 abc defa 并敲回车后，变量 var1 和 vae2 就被分别赋值 abc 和 def 了
```

④ 参数置换方式为变量赋值

`${变量: 一字串}` 如果变量被设定并非空，则返回是变量的值，否则是字符串的值。
`${变量: +字串}` 如果变量被设定并非空，则返回是字串的值，否则是变量的值（即空值）。
`${变量: =字串}` 如果变量被设定并非空，则返回变量的值，否则是字符串的值，同时变量被设成字符串。
`${变量: ? 字串}` 如果变量被设定并非空，则返回变量的值，否则返回报错。

(2) 编写测试程序 var.sh，测试观察变量变化效果：

```
#!/bin/sh

var1=abc
var2=${var1:-"hello"}
echo "var1=$var1 var2=$var2"
var3=
var4=${var3:-"hello"}
echo "var3=$var3 var4=$var4"

var5=abc
var6=${var5:+ "hello"}
echo "var5=$var5 var6=$var6"
var8=${var7:+ "hello"}
echo "var7=$var7 var8=$var8"

var9=abc
```

```

var10=${var9:="hello"}
echo "var9="$var9 "var10="$var10
var11=
var12=${var11:="hello"}
echo "var11="$var11 "var12="$var12

var13=abc
var14=${var13:? "hello"}
echo "var13="$var13 "var14="$var14
var16=${var15:? "hello"}
echo "var15xx="$var15"

echo "game over"

```

增加执行权限并执行:

```

$chmod u+x var.sh
$./var.sh

```

执行结果如下:

```

var1=abc var2=abc
var3= var4=hello
var5=abc var6=hello
var7= var8=
var9=abc var10=abc
var11=hello var12=hello
var13=abc var14=abc
./var.sh: line 27: var15: hello

```

4. 位置变量

Shell 脚本可以向脚本命令行传递参数。在 Shell 中, \$0 表示执行的程序名, \$1~\$9 是传递的命令行参数, Shell 脚本最多能传递 9 个参数, \$1~\$9 称为 Shell 内置的位置变量。shift 会让位置参数左移一位, 即 \$2 变 \$1、\$3 变 \$2。

(1) 编写测试程序 arg.sh, 测试位置变量变化效果:

```

#!/bin/sh
echo NO.0 $0
echo NO.1 $1
echo NO.2 $2
echo NO.3 $3
echo NO.4 $4
echo NO.5 $5
echo NO.6 $6
echo NO.7 $7
echo NO.8 $8
echo NO.9 $9

shift
echo shifting
echo NO.0 $0
echo NO.1 $1
echo NO.2 $2

```

```
echo NO.3 $3
echo NO.4 $4
echo NO.5 $5
echo NO.6 $6
echo NO.7 $7
echo NO.8 $8
echo NO.9 $9
```

(2) 增加脚本执行权限并执行:

```
$ chmod u+x arg.sh
$ ./arg.sh arg1 arg2 arg3 arg4
```

(3) 执行结果如下:

```
NO.0 ./arg.sh
NO.1 arg1
NO.2 arg2
NO.3 arg3
NO.4 arg4
NO.5
NO.6
NO.7
NO.8
NO.9
shifting
NO.0 ./arg.sh
NO.1 arg2
NO.2 arg3
NO.3 arg4
NO.4
NO.5
NO.6
NO.7
NO.8
NO.9
```

5. 字符串变量

字符串变量左右应加双引号, 否则会报错。

(1) 编写测试程序 `str.sh`, 测试字符串变量变化效果:

```
#!/bin/sh
string1=good morning  #没有双引号引起字符串, 执行时会报错
string2="good morning"
echo "string1:$string1"
echo "string2:$string2"
unset string2  #清空变量的值
echo "string2:$string2"
```

(2) 执行 `sh str.sh`, 执行结果如下:

```
str.sh: 3: morning: not found
string1:
string2:good morning
string2:
```

6. 表达式求值

`expr` 命令用来对表达式进行求值，其操作符和运算符之前必须有空格隔开。在 Shell 脚本中，数学表达式直接运算需要用两对圆括号括起来。

【例 3-3】 `expr.sh` 脚本。

(1) 编写测试程序 `expr.sh`，测试表达式求值的效果：

```
#!/bin/sh

expr 3 + 9
expr 9 % 2
expr 3 \* 2    #\表示转义

sum=$((3+2))
echo sum:$sum
mod=$(( 3 % 2 ))
echo mod:$mod
mul=$(( 3 * 2 ))
echo mul:$mul

a=3
c=$(( $a + 8 ))
echo c:$c
```

(2) 执行 `sh expr.sh`，执行结果如下：

```
12
1
6
sum:5
mod:1
mul:6
c:11
```

【例 3-4】 `let.sh` 脚本。

在 Shell 中，使用 `let` 内置命令可以完成对数值的运算。

(2) 编写测试程序 `let.sh`，测试 `let` 命令的使用：

```
#!/bin/bash
let a=11
let a=a+5
echo "11 + 5 = $a"

let "a <= 3"          # let "a = a < 3"
echo "\"$a\" (=16) left-shifted 3 places = $a"

let "a /= 4"          # let "a = a / 4"
echo "128 / 4 = $a"

let "a -= 5"          # let "a = a - 5"
echo "32 - 5 = $a"
```



```
let "a *= 10"      # let "a = a * 10"
echo "27 * 10 = $a"

let "a %= 8"       # let "a = a % 8"
echo "270 modulo 8 = $a (270 / 8 = 33, remainder $a)"
exit 0
```

(2) 执行 ./let.sh, 执行结果如下:

```
11 + 5 = 16
"$a" (=16) left-shifted 3 places = 128
128 / 4 = 32
32 - 5 = 27
27 * 10 = 270
270 modulo 8 = 6 (270 / 8 = 33, remainder 6)
```

3.2.4 Shell 脚本执行

Shell 命名规则一般是 filename.sh, 结尾以 .sh 表示文件类型。

Shell 脚本有两种执行方式: 一种为 sh filename.sh, 第二种执行方式是对文件增加执行权限然后敲入执行码进行执行, 如 chmod u+x filename.sh; ./filename1.sh。

3.2.5 Shell 退出状态

1. Shell 退出状态说明

图 3-1 展示了 Shell 脚本的执行流程及退出状态的保存方法。由于 Shell 解释器创建子进程执行 Shell 脚本, 因此 Shell 进程是 Shell 脚本的父进程, 所以 Shell 可以得到子进程的状态, Shell 脚本内命令同理。退出状态必须是十进制数, 范围必须是 0~255, Shell 脚本执行成功返回 0, 报错返回非 0。

\$? 是一个 Shell 中的内置变量, 代表着最后一次运行进程的退出状态码。

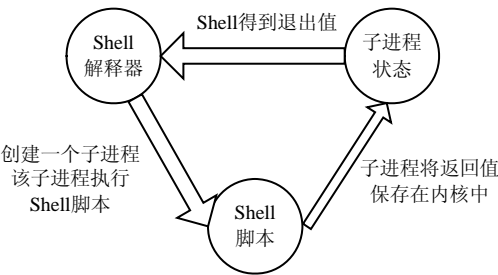


图 3-1 Shell 退出状态图

2. Shell 退出状态举例

下面的用例用来说明 Shell 的退出状态。

(1) 编写测试程序 test.c:

```
#include <stdio.h>
int main(void)
{
    printf("test program\n") ;
    return 35 ;
}
```

(2) 在 Shell 命令行中编译该程序:

```
gcc test.c -o test
```

(3) 编辑 `exit.sh` 脚本, 脚本内容如下:

```
#!/bin/sh
#exit.sh 测试不同进程的退出状态码

./test #执行 test 程序
echo statas=$?

exit 45
```

(4) 执行 `exit.sh` 脚本:

```
sh exit.sh
```

(5) 执行结果如下:

```
test program
statas=35
```

(6) 查看最后一次退出状态:

```
$ echo $?
45
```

3.3 Shell 的输入和输出

3.3.1 Shell 的输入

Shell 用来输入的指令是 `read` 函数, 其格式说明如下:

```
read 变量 1 [变量 2]
```

利用 `read` 函数可以交互式地为变量赋值, 当然也可以通过制表符或空格为多个变量赋值, 使用 `read` 函数读入变量的三种情况说明如下:

- ① 如果变量的个数多于输入字符串的字符个数, 则依次赋值, 剩下的变量取空值。
- ② 如果变量的个数等于输入字符串的字符个数, 则一一对应赋值。
- ③ 如果变量的个数小于输入字符串的字符个数, 则除依次赋值外, 最后一个变量接纳剩下的所有字符串。

【例 3-5】 read 函数的例子

read.sh 源代码如下：

```
#!/bin/sh
echo "input your name and age:"
read name age
echo " name is :"$name
echo "age is :"$age
```

上述代码的意思是从终端中输入两个值，分别赋值给 name 和 age 这两个变量，然后利用 echo 函数将其输出，执行结果如下：

```
$sh read.sh
input your name and age:
sky 3000 --键盘输入
name is :sky
age is :3000
```

3.3.2 Shell 的输出

1. echo 函数介绍

echo 是 Shell 中实现文本和变量输出的函数，能够输出提示信息，显示执行结果和报告执行状态等。

echo 函数后面的各参数之间以空格隔开，以换行符终止。如果数据之间须保留多个空格，则要用双引号把它们整个给引起来，以便 Shell 对它们进行正确的操作。

echo 函数中，还定义了一组转义字符，用于输出控制或打印无法显示的字符。在使用转义字符的时候，要加入“-e”选项。

2. 输出转义字符

表 3-2 列出了 Shell 中的转义字符并对其作用进行了说明，echo 函数利用这些转义字符，可以打印出无法显示的字符。

表 3-2 转义字符及其作用说明表

转义字符	作 用
\a	响铃报警
\b	后退一个字符位置
\c	它出现在参数的最后位置。在它之前的参数被显示后，光标不换行，新的输出信息接在本行后
\e	转义字符
\f	换页
\n	显示换行
\r	回车
\t	制表符
\v	垂直制表符
\\	反斜线本身

3.4 Shell 测试条件

Shell 提供两种测试条件的方式，利用 `test` 命令和利用方括号形式，其定义格式如下：

```
test -d $dir
[-d $dir]
```

这两种方式完全等价，即 `[表达式]` 等价于 `test 表达式`。

1. 条件测试分类

条件测试可以分为四类：字符串测试、数值测试、逻辑测试、文件属性测试。

2. 字符串测试

表 3-3 列出了对字符串测试操作的说明，字符串测试的作用是测试字符串操作的返回值。注意使用 `=`、`!=`、`<`、`>` 这些符号时，两边需要加空格。

表 3-3 字符串测试表

参 数	作 用
-z s1	如果字符串 s1 的长度为 0，则测试条件为真
-n s1	如果字符串 s1 的长度大于 0，则测试条件为真
s1	如果字符串 s1 不是空字符串，则测试条件为真
s1 = s2	如果字符串 s1 等于字符串 s2，则测试条件为真
s1 != s2	如果字符串 s1 不等于字符串 s2，则测试条件为真
s1 < s2	如果按字符顺序，字符串 s1 在字符串 s2 之前，则测试条件为真
s1 > s2	如果按字符顺序，字符串 s2 在字符串 s1 之前，则测试条件为真

下面以 `test_str1.sh`、`test_str2.sh` 两个用例来说明字符串测试的使用方法。

【例 3-6】 `test_str1.sh` 用例。

(1) 编写测试程序 `test_str1.sh`:

```
#!/bin/sh
echo please input name:
read name
if test $name
then
    echo "name is :"$name
else
    echo "name is null"
fi
```

(2) 执行 `sh test_str1.sh`，执行结果如下：

```
please input name:          --直接回车

name is null
```

【例 3-7】 test_str2.sh 用例。

(1) 编写测试程序 test_str2.sh:

```
#!/bin/sh

str1="happy"
str2="happy"
str3=

#测试 str1 与 str2 相等
test $str1 = $str2
echo $?

#测试 str3 是否是空串
test -z $str3
echo $?

#测试 str1 与 str2 不相等
test $str1 != $str2
echo $?

#使用 test 另一种方式[]来实现上述三种判断
echo "using [ ] "

#测试 str1 与 str2 相等
[ $str1 = $str2 ]
echo $?

#测试 str3 是否是空串
[ -z $str3 ]
echo $?

#测试 str1 与 str2 不相等
[ $str1 != $str2 ]
echo $?
```

(2) 执行 sh test_str2.sh, 执行结果如下:

```
0
0
1
using [ ]
0
0
1
```

请大家扩充 Shell 脚本功能, 把所有参数使用一遍, 进行举一反三, 以下同。

3. 数值测试

表 3-4 列出了数值测试的使用方法, 数值测试主要用于两个数值之间大小的比较。

表 3-4 数值测试表

参 数	作 用
n1 -eq n2	如果整数 n1 等于 n2，则测试条件为真
n1 -ne n2	如果整数 n1 不等于 n2，则测试条件为真
n1 -lt n2	如果整数 n1 小于 n2，则测试条件为真
n1 -le n2	如果整数 n1 小于或等于 n2，则测试条件为真
n1 -gt n2	如果整数 n1 大于 n2，则测试条件为真
n1 -ge n2	如果整数 n1 大于或等于 n2，则测试条件为真

下面以 test_number.sh 用例来说明数值的使用方法。

【例 3-8】 test_number.sh 用例。

(1) 编写测试程序 test_number.sh:

```
#!/bin/sh

a=1
b=3

test $a -eq $b
echo $?

test 6 -eq 6
echo $?

#使用[]方式完成上述功能
echo "using [ ] "
[ $a -eq $b ]
echo $?

[ 6 -eq 6 ]
echo $?
```

(2) 执行 sh test_number.sh，执行结果如下：

```
1
0
using [ ]
1
0
```

4. 逻辑测试

表 3-5 列出了逻辑测试的使用方法。逻辑运算符的作用是进行逻辑语句的判断，也就是对“与”、“或”、“非”条件的判断。逻辑表达式中优先级的顺序是：“()”运算符>“!”运算符>“-a”运算符>“-o”运算符。

表 3-5 逻辑测试表

参 数	作 用
!	逻辑“非”，放在任意逻辑表达式之前，使原来为真的表达式变为假，使原来为假的表达式变为真
-a	逻辑“与”，放在两个逻辑表达式中间，只有两个表达式都为真，结果才为真，否则为假

续表

参 数	作 用
-o	逻辑“或”，放在两个逻辑表达式中间，只有两个表达式都为假，结果才为假，否则为真
()	圆括号可以把一个逻辑表达式括起来，使之成为一个整体，优先得到运算

下面以 test_log.sh 用例来说明逻辑测试的使用方法。

【例 3-9】 test_log.sh 用例。

(1) 编写测试脚本 test_log.sh:

```
#!/bin/sh

[ -x $1 -a $0 ] #检查两个文件是否同时可执行，其中一个是 Shell 脚本本身
echo $?

[ -w $1 -o $0 ] #检查两个文件是否有一个可写，其中一个是 Shell 脚本本身
echo $?
```

(2) 增加脚本执行权限:

```
$chmod +x test_log.sh
```

(3) 使用 ls 查看文件参数:

```
$ls test_log.sh test.c
-rw-rw-r-- 1 zfb zfb 86 12月 23 10:11 test.c
-rwxrwxr-x 1 zfb zfb 164 12月 23 19:25 test_log.sh
```

(4) 执行 ./test_log.sh test.c，执行结果如下:

```
$. /test_log.sh test.c
1
0
```

5. 文件属性测试

表 3-6 列出了文件属性测试的使用方法，文件属性测试用于测试文件类型。

表 3-6 文件属性测试表

参 数	作 用
-r 文件名	若文件存在并且是用户可读的，则测试条件为真
-w 文件名	若文件存在并且是用户可写的，则测试条件为真
-x 文件名	若文件存在并且是用户可执行的，则测试条件为真
-f 文件名	若文件存在并且是普通文件，则测试条件为真
-d 文件名	若文件存在并且是目录文件，则测试条件为真
-p 文件名	若文件存在并且是命名的 FIFO 文件，则测试条件为真
-b 文件名	若文件存在并且是块设备文件，则测试条件为真
-c 文件名	若文件存在并且是字符设备文件，则测试条件为真
-s 文件名	若文件存在并且文件的长度大于 0，则测试条件为真
-t 文件描述字	若文件被打开并且文件描述字是与终端设备相关的，则测试条件为真。默认的“文件描述字”是 1

下面以 `test_file.sh` 用例来说明文件属性测试的使用方法。

【例 3-10】 `test_file.sh` 用例。

(1) 编写测试脚本 `test_file.sh`:

```
#!/bin/sh

file=test.c

[ -r $file ] #测试读权限
echo $?
[ -w $file ] #测试写权限
echo $?
[ -x $file ] #测试执行权限
echo $?
[ -f $file ] #测试是否是文件
echo $?
[ -d $file ] #测试是否是目录
echo $?
```

(2) 增加脚本执行权限:

```
$chmod +x test_file.sh
```

(3) 使用 `ls` 查看 `test.c` 文件:

```
$ls test_log.sh test.c
-rw-rw-r-- 1 zfb zfb 86 12月 23 10:11 test.c
```

(4) 使用 `./test_file.sh` 查看执行结果:

```
0
0
1
0
1
```

3.5 Shell 的流程控制结构

本节介绍 Shell 流程控制语句, Shell 流程控制语句有 `if` 语句、`case` 语句、`while` 语句、`until` 语句、`for` 语句和跳转语句 (`break`、`continue`、`exit`), 下文将介绍这些语句的语法形式和使用方法。

3.5.1 if 语句

`if` 语句条件返回值为 0 表示条件测试为真; 如果条件命令执行不成功, 其返回值不等于 0, 条件测试就为假。

`if` 语句的语法形式如下:


```
if 测试条件1
then 命令或命令表
elif 测试条件2
then 命令或命令表
else 命令或命令表
fi
```

其中,elif 部分和 else 部分可以省去,一种结构可以演化为三种结构。下面以 eq_str.sh、file_test.sh 两个用例来说明 if 语句的使用方法。

【例 3-11】 eq_str.sh 用例。

if 语句条件为文件属性测试和字符串测试时需要使用“if [[文件属性测试或字符串测试]]”这种格式。

(1) 编写测试脚本 eq_str.sh:

```
str1="happy new year"
str2="happy new year"

if [[ $str1 = $str2 ]]
then
    echo "they are equal"
fi
```

(2) 增加脚本执行权限:

```
$chmod u+x eq_str.sh
```

(3) 执行 ./eq_str.sh, 执行结果如下:

```
they are equal
```

【例 3-12】 file_test.sh 用例。

(1) 编写测试脚本 file_test.sh:

```
#!/bin/sh
if [ -f $1 ] #普通文件
then
    echo "regular file"
elif [ -d $1 ] #目录文件
then
    echo "dir file" #链接文件
elif [ -l $1 ]
then
    echo "symlink file"
fi
```

(2) 增加脚本执行权限:

```
$chmod u+x file_test.sh
```

(3) 执行 ./file_test.sh, 执行结果如下:

regular file

3.5.2 case 语句

case 语句是一种多重判断语句，类似于多个 if-elif 操作。case 语句的执行原理，是将字符串与各个模式顺次匹配，若满足条件则执行，否则继续查找，如果没有匹配成功的，则不执行任何语句，直接退出。

使用 case 语句时，应注意以下事项：

- ① 每个模式匹配后的处理语句，是以“;;”两个分号进行结束。
- ② 模式串表达式应该有唯一性，不要出现几个模式串表达式能够相互转换的情况，这样不利于语句调试。
- ③ 一个模式表达式可以包含多个模式串，但要用“|”隔开，“|”在这里是“或”的关系。

case 语句是一个基于模式匹配的多路分支结构，其一般语法形式如下：

```
case word in
pattern 1) 命令表 1;;
pattern 2) 命令表 2;;
...
*) 缺省命令表;;
esac
```

下面以 case.sh、case_menu.sh 两个用例来说明 case 语句的使用方法。

【例 3-13】 case.sh 用例。

(1) 编写测试脚本 case.sh:

```
#!/bin/sh
echo please input your name:
read name
case $name in
Tom)
    echo your name is tom ;;
Jim)
    echo your name is Jim ;;
*)
    echo "sorry we don't know your name" ;;
esac
```

(2) 执行 sh case.sh，执行结果如下：

```
please input your name:
Jim --输入
your name is Jim
```

【例 3-14】 case_menu.sh 用例。

(1) 编写脚本 case_menu.sh:

```
#!/bin/sh
echo "1 save"
echo "2 load"
echo "3 exit"
echo #输出一个换行
echo "please input chioce"
read chioce
#快捷键如下
#s--存储(save)
#l--加载(load)
#e--退出(exit)
case $chioce in
    1 | S | s)
        echo "save";;
    2 | L | l)
        echo "load";;
    3 | $ | s)
        echo "exit";;
    *) #其他的输入情况
        echo "invalid choice"
        exit 1;;
esac
exit 0
```

(2) 执行 `sh case_menu.sh`, 执行结果如下:

```
1 save
2 load
3 exit

please input chioce
s
save
```

3.5.3 while 语句

`while` 语句的执行过程是: 先测试条件语句是否为真, 若为真, 则执行循环体, 当执行完当前命令后, 再进行条件测试, 直到条件结果为假, 循环结束。

这里的条件测试语句既可以是 `test` 语句, 也可以是运行命令的返回值, 若返回值大于 0, 则表示条件为真, 否则条件为假。

`while` 语句的语法形式如下:

```
while 测试条件
do
    命令表
done
```

下面以 `ls_file.sh`、`read_while.sh` 两个用例来说明 `while` 语句的使用方法。

【例 3-15】 `ls_file.sh` 用例。

执行时每隔 1 秒显示 test.c 文件大小，使用 Ctrl+C 结束脚本执行。

```
#!/bin/sh
while true
do
    sleep 1 ;
    ls -l test.c
done
```

【例 3-16】 read_while.sh 用例。

(1) 编写测试脚本 read_while.sh:

```
#!/bin/sh
type="";
echo input your type:
read type
while [ $type != "quit" ]
do
    echo "your input is :"$type
    echo input your type:
    read type
done
```

(2) 执行 sh read_while.sh, 执行结果如下:

```
input your type:
why
your input is :why
input your type:
quit
```

3.5.4 until 语句

until 语句在形式上是 while 语句的一种变形。对于 until 语句中的条件测试语句，如果条件为假，则执行，否则不执行。

until 语句的语法形式如下：

```
until 测试条件
do
    命令表
done
```

下面以 until.sh 用例来说明 until 语句的使用方法。

【例 3-17】 until.sh 用例。

(1) 编写测试脚本 until.sh:

```
#!/bin/sh
type="";
echo input your type:
read type
```

```
until [ $type = "quit" ]
do
    echo "your input is :"$type
    echo input your type:
    read type
done
```

(2) 执行 `sh until.sh`, 执行结果如下:

```
input your type:
what
your input is :what
input your type:
quit
```

3.5.5 for 语句

1. for 语句统一语法形式

for 语句是 Shell 中经常使用的循环语句, for 语句统一的语法形式如下:

```
for 变量名 in 循环参数列表
do
    命令表
done
```

上面统一的语法形式, 可以演化成下面三种表现形式。

2. 数组作为循环参数

for 语句循环参数是数组表时, 语法形式如下:

```
for 变量名 in 数组表
do
    命令表
done
```

下面以 `for_array.sh` 用例说明 for 语句循环参数是数组表时的用法。

【例 3-18】 `for_array.sh` 用例。

(1) 编写测试脚本 `for_array.sh`:

```
#!/bin/sh

for word in Hello to you
do
    echo $word
done

array="what who where"
for word in $array
do
    echo $word
done
```

```
file=`ls`  
for word in $file  
do  
    echo $word  
done
```

(2) 执行 `sh for_array.sh`, 执行结果如下:

```
Hello  
to  
you  
what  
who  
where  
args.sh  
case_menu.sh
```

3. 以正则表达式作为循环参数

`for` 语句循环参数是正则表达式时, 语法形式如下:

```
for 变量 in 正则表达式  
do  
    命令表  
done
```

下面以 `for_regular.sh` 用例说明 `for` 语句循环参数是正则表达式时的用法。

【例 3-19】 `for_regular.sh` 用例。

(1) 编写测试脚本 `for_regular.sh`:

```
#!/bin/sh  
for file in /dev/tty[1-3]  
do  
    ls -l $file  
done
```

(2) 执行 `sh for_regular.sh`, 执行结果如下:

```
crw----- 1 root root 4, 1 12月 23 08:24 /dev/tty1  
crw----- 1 root root 4, 2 12月 23 08:24 /dev/tty2  
crw----- 1 root root 4, 3 12月 23 08:24 /dev/tty3
```

4. 位置参数方式作为循环参数

`for` 语句循环参数是位置参数时, 语法形式如下:

```
for 变量 in $*  
do  
    命令表  
done
```

下面以 `for_args.sh` 用例说明 `for` 语句循环参数是位置参数时的用法。

【例 3-20】 for_args.sh 用例。

(1) 编写测试脚本 for_args.sh:

```
#!/bin/sh
for word in $*
do
    echo $word
done
```

(2) 执行 sh for_args.sh arg1 arg2 arg3, 执行结果如下:

```
arg1
arg2
arg3
```

3.5.6 跳转语句

1. break 语句

break 语句是一个退出循环的命令, 主要用于多层循环的嵌套, 它的一般使用形式为:

```
break [n]
```

其中, n 用来表示跳出几层循环, 默认值为 1, 即退出本次循环。

2. continue 语句

continue 语句与 break 语句有相同之处, 都用于终止本次循环, 区别在于 break 语句是退出整个循环, 即不再执行剩下的循环操作, 而 continue 语句是停止本次循环体的执行, 转向循环体中的下一次循环。

continue 语句的语法为:

```
continue [n]
```

其中, n 用来表示跳出几次循环, 默认值为 1。

3. exit 语句

exit 语句是退出正在执行的 Shell 脚本, 可以主动指定返回值。

exit 语句的语法为:

```
exit [n]
```

其中, n 是主动设定的返回值, 如果未显式给定 n 的值, 则该值默认取最后一次命令的执行状态作为返回值。

3.6 Shell 数组

在 Shell 中可以使用数组来存储同类型的数值集合。一般 Shell 中支持一维数组, 但不限

定数组的具体大小，数组的使用方式是采用指定下标。数组中的下标往往是由 0 开始编号的，一般可以采用直接给出下标的方式，也可以通过算术表达式的方式指定下标。但不管采用哪种方式，都要记住一点，不能给出一个小于 0，或大于数组已存元素个数的值，不然会产生越界的错误。

在数组的操作中，取值的一般方式是：

```
${数组名[下标值]}
```

相对应的数组的赋值操作的一般方式是：

```
数组名[下标值]=值
```

对于数组的赋值，可以采用单个元素逐一进行赋值的方式，也可以采用一次性赋值的方式，但要注意，一次性赋值时，值与值之间要用空格隔开，赋值方法如下：

```
数组名=(value1 value2 value3...)
```

在数组中可以使用 * 或 @ 符号来代替下标，这里 * 或 @ 就是上文说明的通配符。

下面以 array.sh 用例来说明 Shell 数组的使用方法。

【例 3-21】 array.sh 用例。

(1) 编写测试脚本 array.sh:

```
name=(tom jim jane test1)
echo "name[0] is :${name[0]}"
echo "name[1] is :${name[1]}"
echo "name[2] is :${name[2]}"
echo "name[3] is :${name[3]}"
echo "name set is :${name[*]}"
echo "name set is :${name[@]}"
```

(2) 增加脚本执行权限:

```
$chmod u+x array.sh
```

(3) 执行 ./array.sh，执行结果如下:

```
name[0] is :tom
name[1] is :jim
name[2] is :jane
name[3] is :test1
name set is :tom jim jane test1
name set is :tom jim jane test1
```

3.7 Shell 函数

函数代表一种模式化的设计思想，可以将一些常用的、内聚度较高的操作，封装成函数，在需要时进行调用。

在 Shell 中可以按照一定的规则，定义一组命令集，组成一个过程，这个过程有过程名，在

执行这个命令集时，只需要通过使用这个过程名，就能实现执行过程中相关命令的功能，这个过程就叫函数。

函数其实是一个模块化的概念，按一定功能，设定一个模块，在使用时，只须指定模块名，便能达到操作的目的。Shell 中的函数一次定义，可以多次使用。执行函数操作，并不需要创建新进程，而是在当前的 Shell 进程中运行。

1. Shell 函数定义原型

Shell 中函数定义方法如下：

```
function 函数名()  
{  
    语句  
}
```

在这里，关键字 `function` 是可以不显式指定的。在使用函数时，要注意函数应先定义再使用，调用函数时，只须指定函数名，不用加后面的 `()`。

下面以 `show.sh` 用例来说明 Shell 函数的使用方法。

【例 3-22】 `show.sh` 用例。

(1) 编写测试脚本 `show.sh`：

```
show()  
{  
    echo $a $b $c  
    echo $1 $2 $3  
}  
a=111  
b=222  
c=333  
d=444  
f=555  
e=666  
echo "Function Begin"  
show $d $e $f  
echo "Function Finished"
```

(2) 执行 `sh show.sh`，执行结果如下：

```
Function Begin  
111 222 333  
444 666 555  
Function Finished
```

2. Shell 函数的参数与返回值

① 变量传递的两种方法

变量传递有两种方法：其一为变量直接传递法，数量不限，如上文 `show.sh` 中变量 `a`、`b`、`c`；其二为位置参数法，数量最多 9 个，如上文 `show.sh` 中 `$1`、`$2`、`$3`。

② 函数返回值

函数执行到最后一条语句后就会退出，但也可以主动调用 `return` 语句来实现提前退出。`return` 语句的使用方式相对简单，原型为：`return n`，其中，`n` 的值可以主动指定，若采用默认的方式，则退出值为最近一个命令的退出码。

3.8 I/O 重定向

1. Linux 文件描述

“一切皆是文件”是 UNIX/Linux 的基本设计哲学之一。不仅普通的文件，目录、字符设备、块设备、套接字等在 UNIX/Linux 中都是以文件的形式被对待的。它们虽然类型不同，但是对其提供的却是同一套操作界面。

在 Linux 中每一个进程都由 `task_struct`（参见图 12-4 进程 `task_struct` 文件结构）数据结构来定义。`task_struct` 数据结构的 `files` 选项指向打开文件描述符。每一个进程默认打开三个文件。这三个文件描述符是 `files_struct` 数据结构中的 `fd[0]`、`fd[1]` 和 `fd[2]`，即文件描述符 0、1、2，分别指向标准输入（键盘）、标准输出（屏幕）、标准错误（屏幕）。

在 Linux 系统中，文件描述符（File Descriptor）是用一个数字来表示。表 3-7 列出了进程默认打开的文件描述符表。

表 3-7 进程默认打开文件描述符表

文件描述符	名 称	常用缩写	默认指向
0	标准输入	<code>stdin</code>	键盘
1	标准输出	<code>stdout</code>	屏幕
2	标准错误输出	<code>stderr</code>	屏幕

2. 基本 I/O 重定向

I/O 重定向表示重新定位数据的流向，下面说明的是常见的 I/O 重定向方法。

- `cmd > file:` 把 `stdout` 重定向到 `file` 文件中。
- `cmd >> file:` 把 `stdout` 重定向到 `file` 文件中（追加）。
- `cmd 1> file:` 把 `stdout` 重定向到 `file` 文件中。
- `cmd > file 2>&1:` 把 `stdout` 和 `stderr` 一起重定向到 `file` 文件中。
- `cmd 2> file:` 把 `stderr` 重定向到 `file` 文件中。
- `cmd 2>> file:` 把 `stderr` 重定向到 `file` 文件中（追加）。
- `cmd >> file 2>&1:` 把 `stdout` 和 `stderr` 一起重定向到 `file` 文件中（追加）。

`cmd <file>file2:` `cmd` 命令以 `file` 文件作为 `stdin`, 以 `file2` 文件作为 `stdout`。

`cmd < file:` 以 `file` 文件作为 `stdin`。

`cmd << delimiter:` 从 `stdin` 中读入, 直至遇到 `delimiter` 分界符。

`cmd <<- delimiter:` 从 `stdin` 中读入, 直至遇到 `delimiter` 分界符, 输入去掉行首 `Tab` 键。

3. 高级 I/O 重定向

下面说明复杂 I/O 重定向的方法, 较少使用, 了解即可。

`>&n:` 使用系统调用 `dup(2)` 复制文件描述符 `n`, 并把结果用做标准输出。

`<&n:` 标准输入复制自文件描述符 `n`。

`<&-:` 关闭标准输入 (键盘)。

`>&-:` 关闭标准输出。

`n<&-:` 表示将文件描述符 `n` 输入关闭。

`n>&-:` 表示将文件描述符 `n` 输出关闭。

`cmd 2>file:` 运行一个命令并把错误输出 (文件描述符 2) 定向到 `file`。

`cmd 2>&1:` 运行一个命令并把它标准输出和输入合并 (严格地说是通过复制文件描述符 1 来建立文件描述符 2, 但效果通常是合并了两个流)。

`exec 1>outfilename:` 打开文件 `outfilename` 作为 `stdout`。

`exec 2>errfilename:` 打开文件 `errfilename` 作为 `stderr`。

`exec 0<&-:` 关闭 `fd0`。

`exec 1>&-:` 关闭 `fd1`。

`exec 5>&-:` 关闭 `fd5`。

3.9 Shell 内置命令

1. 内置命令列表

表 3-8 列出了 Shell 的内置命令及其说明, Shell 的内置命令是在 Shell 脚本中能使用的命令。

表 3-8 Shell 内置命令列表

命 令	含 义
:	空命令，返回退出状态零
.	在当前进程的环境下执行程序
break	跳出最内层的循环
break [n]	循环控制命令
alias	为存在的命令列出并创建别名
bg	将一个作业放到后台
bind	显示当前键和函数的绑定，或将键和一个 readline 函数或宏绑定
echo [args]	显示用换行符终止的参数
enable	开启和关闭 Shell 内置命令
eval [args]	读参数作为 Shell 的输入，并执行产生的命令
exec command	执行命令来取代当前的 Shell
exit [n]	以状态 n 退出 Shell
export [var]	使 var 能被子 Shell 识别
fc	用于编辑历史命令的历史编辑命令
fg	将后台作业放到前台
getopts	解析并处理命令行选项
hash	控制内部哈希表以更快地搜索命令
help [command]	显示关于内置命令的帮助信息，如果指定命令，将显示该内置命令的详细帮助
history	显示带行号的历史清单
jobs	列出放在后台的作业
kill [-singal process]	发送信号给指定 PID 号或作业号的进程
getopts	用于 Shell 脚本以解析命令行并检查合法的选项
let	用来对算术表达式求值并将算术计算的结果赋给变量
local	用在函数中以限制变量在函数中的作用域
logout	退出登录 Shell
popd	从目录栈中删除项
pushd	往目录栈中添加项
pwd	显示当前工作目录
read [var]	从标准输入读取一行到变量 var
readonly [var]	使变量 var 只读。不能被复位
return [n]	从一个函数返回，n 是返回的退出值
set	设置选项和位置参量
shift [n]	向左移动位置参量 n 次
stop pid	终止 PID 号进程的执行
suspend	暂停当前 Shell 的执行（如果是一个登录 Shell 就不暂停）
test	检查文件类型且测试条件表达式
times	为从该 Shell 运行的进程显示所累积的用户和系统时间
trap [arg] [n]	当 Shell 接收到信号 n（0、1、2 或 15）时执行参数
type [command]	打印命令的类型。例如，pwd 是一个内置 Shell 命令
typeset	和 declare 一样，设置变量并给它们属性

续表

命 令	含 义
ulimit	显示并设置进程资源限度
umask [octal digits]	设置创建文件时关于文件属主、属组和其他用户执行权限的掩码
unalias	删除别名
unset [name]	删除变量值或函数
wait [pid#n]	等待后台 PID 号为 n 的进程返回并报告终止状态
date	显示时间和时间设置

2. trap 命令

trap 命令用于指定在接收到信号后将要采取的行动，trap 命令的一种常见用途是忽略某些信号或在脚本程序被信号中断时完成清理工作。历史上，Shell 总是使用数字来代表信号，现在提倡使用信号的名字，使用信号名时需要省略 SIG 前缀。在命令提示符下输入命令 trap -l 可以查看信号编号及其关联的名称。

“信号”是指那些被异步发送到一个程序的事件。默认情况下，它们通常会终止一个程序的运行。

trap 命令使用格式如下：

```
trap 'command' signal-list
```

其中 trap 命令的参数分为两部分，前一部分是接收到指定信号时将要采取的行动，后一部分是要处理的信号名。

(1) trap 捕捉到信号之后，可以有以下三种反应方式。

- ① 执行一段程序来处理这一信号。
- ② 接受信号的默认操作。
- ③ 忽视这一信号。

(2) trap 对上面三种方式提供了三种基本形式。

① 设置信号的处理方式，使用第一种形式：

```
trap 'command' signal-list
trap "command" signal-list
```

② 恢复信号的默认操作，使用第二种形式：

```
trap signal-list
```

③ 忽略信号，使用第三种形式：

```
trap " " signal-list
```

在第一种形式 trap 命令中 Shell 接收到 signal-list 清单中数值相同的信号时，将执行

引号中的命令串。

使用 `trap` 时有如下注意事项：

- ① 对信号 11（段违例）不能捕捉，因为 Shell 本身需要捕捉该信号去进行内存的转存。
- ② 在捕捉到 `signal-list` 中指定的信号并执行完相应的命令之后，如果这些命令没有将 Shell 程序终止的话，Shell 程序将继续执行收到信号时所执行的命令后面的命令，这样将很容易导致 Shell 程序无法终止。
- ③ 在 `trap` 语句中，单引号和双引号是不同的，当 Shell 程序第一次碰到 `trap` 语句时，将把 `command` 中的命令扫描一遍，此时若 `command` 是用单引号引起来的话，那么 Shell 不会对 `command` 中的变量和命令进行替换。

表 3-9 列出了能被 `trap` 命令捕获的比较重要的信号列表及其说明。

表 3-9 能够被捕获的比较重要信号列表

命 令	含 义
HUP（1）	挂起，通常因终端掉线或用户退出而引发
INT（2）	中断，通常因按下 Ctrl+C 组合键而引发
QUIT（3）	退出，通常因按下 Ctrl+\ 组合键而引发
ABRT（6）	中止，通常因某些严重的执行错误而引发
ALRM（14）	报警，通常用来处理超时
TERM（15）	终止，通常在系统关机时发送

通常需要忽略的信号有四个，即 HUP、INT、QUIT、TSTP，也就是信号 1、2、3、24，使用下面的语句可以使这些信号被忽略。

```
trap "" 1 2 3 24 或 trap "" HUP INT QUIT TSTP
```

3. date 命令

`date` 命令的功能是显示和设置系统日期和时间。

`date` 命令语法形式如下，查看时间时格式需要带+号。

```
date [选项] [+格式]
```

`date` 常见的选项说明如下：

- ① `-d datestr, --date datestr` 显示由 `datestr` 描述的日期。
- ② `-s datestr, --set datestr` 设置 `datestr` 描述的日期。
- ③ `-u, --universal` 显示或设置通用时间。

表 3-10 列出 `date` 设置和显示时间时格式选项的种类及其说明。这里需要说明的是，只有超级用户才能用 `date` 命令设置时间，一般用户只能用 `date` 查看时间。

表 3-10 date 时间格式说明表

格 式	说 明
%H	小时（00...23）
%I	小时（01...12）
%k	小时（0...23）
%l	小时（1...12）
%M	分（00...59）
%P	显示出 AM 或 PM
%r	时间（hh: mm: ss AM 或 PM），12 小时
%s	从 1970 年 1 月 1 日 00: 00: 00 到目前经历的秒数
%S	秒（00...59）
%T	时间（24 小时制）（hh:mm:ss）
%X	显示时间的格式（%H:%M:%S）
%Z	时区 日期域
%a	星期几的简称（Sun...Sat）
%A	星期几的全称（Sunday...Saturday）
%b	月的简称（Jan...Dec）
%B	月的全称（January...December）
%c	日期和时间（Mon Nov 8 14: 12: 46 CST 1999）
%d	一个月的第几天（01..31）
%D	日期（mm / dd / yy）
%h	和%b 选项相同
%j	一年的第几天（001...366）
%m	月（01...12）
%w	一个星期的第几天（0 代表星期天）
%W	一年的第几个星期（00..53，星期一为第一天）
%x	显示日期的格式（mm/dd/yy）
%y	年的最后两个数字（1999 则是 99）
%Y	年（如 1970、1996 等）

例 1：用指定的格式显示时间。

```
$ date '+This date now is =>%x , time is now =>%X , thank you !'
This date now is =>11/12/99 , time is now =>17:53:01 , thank you !
```

例 2：用默认格式显示当前的时间。

```
# date
Fri Nov 26 15: 20: 18 CST 1999
```

例 3：设置时间为下午 14 点 36 分。

```
# date -s 14:36:00
Fri Nov 26 14: 15: 00 CST 1999
```

例 4：设置时间为 1999 年 11 月 28 号。

```
# date -s 991128
Sun Nov 28 00: 00: 00 CST 1999
```

例 5：按格式显示下一天的时间。

```
# date -d next-day +%Y%m%d
20060328
```

3.10 实用 Shell 脚本

下面三个 Shell 脚本是作者从业中经常使用到的脚本。file_mod.sh 是去掉文件中的回车符，当 Windows 下的文本文件传到 Linux 下时，文件每行会多一个回车符，利用 file_mod.sh 可以批量修改文件，去掉文件尾的回车符；stopproc.sh 脚本是根据进程名杀死一个或多个进程的脚本，此脚本相当于实现 Linux 下的 pkill 命令的功能；backup.sh 脚本是实用数据备份脚本，这是一个技术含量很高而且非常实用的脚本，此脚本实现了按需备份（排除不需要备份的文件）和自动 ftp 传输功能。

1. 去掉文件中回车符

【例 3-23】 file_mod.sh 用例。

```
file="aa.txt aaa.txt"
echo $file

# ^[表示 ESC 键，需要先输 ctrl+v，再按 ESC 键
# ^M 表示回车，需要先输 ctrl+v，再按 M 键
for filename in $file
do
    echo $filename
    vi $filename<<-EOF
    :s/aaa/AAA/g^M^[
    :wq^M^[
    EOF
done
```

2. 根据进程名 kill 进程

【例 3-24】 stopproc.sh 用例。

```
#!/bin/sh

if [ $# -lt 1 ]
then
    echo "usage ./simstop.sh proc_name"
    exit 1
fi

PROCESS=`ps -ef|grep $1|grep -v grep|grep -v PPID|awk '{ print $2}'`
for i in $PROCESS
do
    echo "Kill the $1 process [ $i ]"
    kill -9 $i
done
```


3. 实用数据备份脚本

【例 3-25】 backup.sh 用例。

```
#!/bin/sh
FAPWORKDIR=/home/bep

cd ${FAPWORKDIR}
DATE=`date +%Y%m%d`

rm -f backup/bep.$DATE.tar backup/exclude.list

# 产生排除文件列表
# *.tar *.Z *.gz *.rar *.o .* 等文件都不备份
find -L include -name '*.tar' -o -name '*.Z' -o -name '*.gz' -o -name '*.rar' -o
-name '.*' >> backup/exclude.list
find -L src -name '*.tar' -o -name '*.Z' -o -name '*.gz' -o -name '*.rar' -o
-name '.*' -o -name '*.o' >> backup/exclude.list

tar -chvf backup/bep.$DATE.tar -X backup/exclude.list .profile include src scr
ipts lib
compress -F backup/bep.$DATE.tar

ftp -n 192.168.31.125 <<!
user bepbak bepbak
lcd backup
cd 110_bak
bin
put bep.$DATE.tar.Z
bye
!

rm -f backup/bep.$DATE.tar backup/bep.$DATE.tar.Z backup/exclude.list
```

第 2 篇

Linux C 语言程序设计

- ❖ 第 4 章 C 语言基础
- ❖ 第 5 章 C 语言函数
- ❖ 第 6 章 C 语言数组、结构体及指针
- ❖ 第 7 章 C 语言预处理
- ❖ 第 8 章 格式化 I/O 函数
- ❖ 第 9 章 字符串和内存操作函数
- ❖ 第 10 章 标准 I/O 文件编程
- ❖ 第 11 章 Linux C 语言开发工具 (vi 与 vim、gcc、Makefile、gdb)

学海聆听：

- 凡事预则立，不预则废。
- 万变不离其中，透过现象看本质。
- 天生我材必有用。
- 形而上者谓之道，形而下者谓之器。
- 勤能补拙，熟能生巧；光说不练，枉学百年。
- 择善人而交，择善书而读，择善言而听，择善行而从。
- 博学而笃志，切问而近思。
- 操千曲而后晓声，观千剑而后识器。
- 发现问题，提出问题，分析问题，解决问题，总结问题。
- 大道至简，道法自然。
- 天行健，君子以自强不息；地势坤，君子以厚德载物。
- 天道酬勤，身体是革命的本钱。

第 4 章

C语言基础

C 语言是计算机编程中最典型、最常用的语言，如操作系统、数据库、通信软件和金融、电力等行业软件都是用 C 语言编的。C++语言是 C 语言的扩展，而 Java 语言许多语法又与 C 语言类似。所以作者认为，学好计算机请首先学好 C 语言，C 语言是计算机软件行业的地基，学好 C 语言可以对其他计算机语言触类旁通、一通百通，C 语言，编程世界的王者。学好 C 语言，需要理解和掌握指针，理解了指针就一定程度上理解了 C 语言，掌握好指针是晋升 Linux C 高级程序员的必要条件。

4.1 C 语言基本概念

计算机语言与人类语言一样，都是一种交流的工具。人类语言是人与人之间交流的工具，计算机语言是人与计算机之间交流的工具。所有语言都有它的语法、语素和语用，都有它的语法规则，这样才能被交流的双方相互理解。计算机语言也不例外，我们编写的计算机程序也必须遵守一定的语法规则，才能被编译器所识别，最后翻译成能被 CPU 理解并执行的机器语言，其中机器语言是 CPU 厂商设计的。

1. 计算机语言相关概念解释

① 编译程序：编译程序又称为编译器，是一个语言翻译程序，它把源语言翻译成目标语言。源语言主要指各种计算机语言，目标语言主要指 CPU 能识别的机器码。例如，英语翻译成汉语的过程中，翻译官相当于编译器。

② 编译：编译源语言，生成目标代码并形成机器码的过程，相当于现实翻译中的笔译。C 语言属编译型语言。

③ 解释：把源语言解释成机器码边执行的过程，相当于现实翻译中的口译。Shell 脚本属解释型语言。

④ 编译阶段：编译阶段包括词法分析、语法分析、语义分析、中间代码生成、代码优化、目标代码生成六个阶段，贯穿始终的功能模块有表格管理和出错管理。

⑤ 为什么需要编译程序：正如我们的母语是汉语，当我们想与一个埃及人交流时，虽然我们不懂阿拉伯语，但如果有一个自动语言翻译机（或翻译官），双方的交流就能变得流畅。由于 CPU 指令是二进制指令，人们难以在此基础上进行有效编程，所以发明了各种各样的计算机高级语言，然后通过编译器把人们编写的计算机高级语言程序翻译成 CPU 能理解的二进制指令。

⑥ 程序：通常指的是人们编写的计算机语言源代码和编译后的执行码，其中源代码称为源程序，执行码称为执行程序，程序是静态的。程序相当于一个企业行政部门出台的行政管理文件。

⑦ 进程：程序的一次执行过程，进程是动态的。进程相当于企业各部门拿着行政管理文件的执行过程。

⑧ CPU：CPU 主要与内存通信，其主要功能有解释指令、存数据到内存、从内存取数据、数据计算和中断处理，CPU 最突出的功能为解释指令和数据计算。CPU 的工作是从内存中取出指令并完成指令的自动化执行。

⑨ 内存：内存是存放电脑工作数据的空间，断电后数据丢失。内存的最小单位为字节，内存每一个最小单位空间都有其编号（即内存地址），CPU 是通过内存地址访问内存数据的。在内存看来，内存里存放的数据都是平等的，都是一串串的二进制字符，没有类型，也没有含义，是计算机程序为内存数据赋予特别的类型和意义。

⑩ 变量：一段内存空间的抽象，变量类型决定了变量存放内存空间的大小。将计算机语言编译成机器码后，变量相对于内存的操作就转变为对相应内存地址的存取操作。直接存取是一般变量，间接存取是指针变量。C 语言变量原则为先定义，后使用。

⑪ 机器程序：经过编译器编译后形成 CPU 能理解的机器指令。在机器程序中，只有内存地址，没有变量概念，机器程序中所有变量失去意义，变量的操作都会转换成对内存地址的存取操作。机器程序有如下几种类型的操作：CPU 的计算操作、内存空间的申请与释放、CPU 把寄存器数据存到内存、取内存数据到 CPU 寄存器、内存数据从一片空间复制到另一片空间、中断操作等。

⑫ Linux 系统编译方法：gcc（或 cc）a.c（源代码名称）-o a（执行码）。在这里 a.c 和 a 都是程序，其中 a.c 为源程序，a 为可执行程序。在界面上输入 ./a，a 就开始了执行之旅，其执行过程称为进程。

2. C 语言数据类型种类

图 4-1 给出了 C 语言数据类型种类，在 C 语言中只允许使用下面这些数据类型。

3. 32 个关键字

C 语言有如下 32 个关键字，这些关键字由系统定义，不能重新做其他定义。

```
auto    break    case    char    const    continue    default    do    double
else    enum    extern    float    for    goto    int    long    register    return
short    signed    sizeof    static    struct    switch    typedef    unsigned    union
void    volatile    while
```

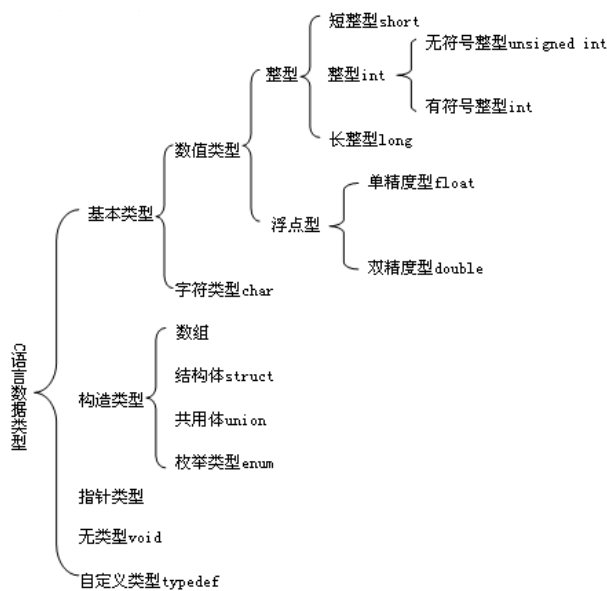


图 4-1 C 语言数据类型种类

4. 9 种控制语句

C 语言控制语句的功能是完成程序流程的控制，C 语言有如下 9 种控制语句。

- ① if()-else (条件语句)
- ② for() (循环语句)
- ③ while() (循环语句)
- ④ do-while() (循环语句)
- ⑤ continue (结束本次循环开始下一次循环语句)
- ⑥ break (跳出 switch 或循环语句)
- ⑦ switch (多分支选择语句)
- ⑧ goto (跳转语句)
- ⑨ return (从函数返回语句)

5. C 程序格式和结构特点

C 语言习惯用小写字母，大小写敏感；不使用行号，无程序行概念；可使用空行和空格；常采用锯齿形书写格式。

下面以一个实例说明 C 语言结构特点。

```
/* example1.1 The first C Program*/  ← 注释
#include <stdio.h>  ← 编译预处理
```

```
main() ← 函数
{
    printf("Hello,World!"); ← 语句
}
```

C 语言由函数、语句和注释三个部分组成，这三部分的说明和要求如下。

(1) 函数与主函数

一个 C 语言源程序可以由一个或多个源文件组成。每个源文件可由一个或多个函数组成。一个源程序不论由多少个文件组成，都有一个且只能有一个 main 函数，即主函数。

源程序中可以有预处理命令（include 命令仅为其中的一种），预处理命令通常应放在源文件头。

每一个说明，每一个语句都必须以分号结尾。但预处理命令，函数头和花括号“}”之后不能加分号。

标识符、关键字之间必须至少加一个空格以示间隔，若已有明显的间隔符，也可不再加空格来间隔。

(2) 程序语句

C 程序由语句组成，用“;”作为语句终止符。

{ } 表示一个语句的整体。if、for、while 包含多条语句时，需要用 { } 括起来表示一个整体，单条语句则可直接用“;”表示语句终止。

(3) 注释

/* */ 为注释，不能嵌套，注释不产生编译代码。

6. C 语言程序的开发过程

编辑 → 编译（预编译、编译、链接）→ 形成执行码 → 执行

7. 书写 C 语言程序时应遵循的规则

一个说明或一个语句应该占一行。

用 { } 括起来的部分，通常表示程序的某一层结构。{ } 一般与该结构语句的第一个字母对齐，并单独占一行。

使用缩进方式编程，低一层次的语句比高一层次的语句缩进若干空格后书写，以便看起来更加清晰，增加程序的可读性。

有足够的注释。有合适的空行。

8. C 语言的字符集

字符是组成语言的最基本的元素。C 语言字符集由字母、数字、空格、标点和特殊字符组成。

在字符串和注释中还可以使用汉字或其他可表示的图形符号。

(1) 字母

小写字母 `a~z` 共 26 个，大写字母 `A~Z` 共 26 个。

(2) 数字

`0~9` 共 10 个。

(3) 空白符

空格符、制表符、换行符等统称为空白符，空白符只在字符常量和字符串中起作用。在其他地方出现时，只起间隔作用，编译程序对它们忽略不计。在程序中适当的地方使用空白符将增加程序的清晰性和可读性。

(4) 标点和特殊字符

包括 “`,`”、“`;`” 等标点和 “`[`”、“`]`”、“`*`” 等特殊字符。

9. C 语言词汇

在 C 语言中使用的词汇分为六类，分别为标识符、关键字、运算符、分隔符、常量和注释符，具体说明如下。

(1) 标识符

在程序中使用的常量名、变量名、函数名等统称为标识符。除库函数的函数名由系统定义外，其余都由用户自定义。C 规定，标识符只能是由字母 (`A~Z`, `a~z`)、数字 (`0~9`)、下画线 (`_`) 组成的字符串，并且其第一个字符必须是字母或下画线。

`a`、`x`、`x3`、`BOOK_1`、`sum5` 这 5 个标识符是合法的标识符。

以下标识符是非法的：

- `3s` 以数字开头；
- `s*T` 出现非法字符 `*`。

(2) 关键字

关键字是由 C 语言规定的具有特定意义的字符串，通常也称为保留字，用户定义的标识符不应与关键字相同。C 语言的关键字分为以下几类：

① 类型说明符

用于定义、说明变量、函数或其他数据结构的类型，如 `int`、`double` 等。

② 语句定义符

用于表示一个语句的功能，如 `if-else` 就是条件语句的语句定义符。

③ 预处理命令字

用于表示一个预处理命令，如 `include`。

(3) 运算符

C 语言中含有相当丰富的运算符。运算符与变量、函数一起组成表达式，表示各种运算功能。运算符由一个或多个字符组成。

(4) 分隔符

在 C 语言中采用的分隔符有逗号和空格两种。逗号主要用在类型说明和函数多个形参中分隔各个变量，空格多用于语句各单词之间作为间隔符。

(5) 常量

C 语言中使用的常量可分为数字常量、字符常量、字符串常量、符号常量、转义字符等多种。

(6) 注释符

C 语言的注释符是以 “`/*`” 开头并以 “`*/`” 结尾的字符串，在 “`/*`” 和 “`*/`” 之间的即为注释。程序编译时，不对注释做任何处理，注释可出现在程序中的任何位置，注释用来向用户提示或解释程序的意义。

10. C 语言算法

C 语言是结构化设计语言，结构化程序的灵魂是算法。瑞士 Niklaus Wirth 教授在 20 世纪 60 年代提出了 “数据结构+算法=程序” 的公式。做任何事情都有一定的步骤，为解决一个问题而采取的方法和步骤，就称为算法。算法具有有穷性、确定性、有零个或多个输入、有一个或多个输出、有效性等特征。C 语言算法是通过结构化设计方法来实现的，其特点为自顶向下、逐步细化、模块化设计和结构化编码。

11. C 语言编译过程

图 4-2 给出了 C 语言的编译流程。

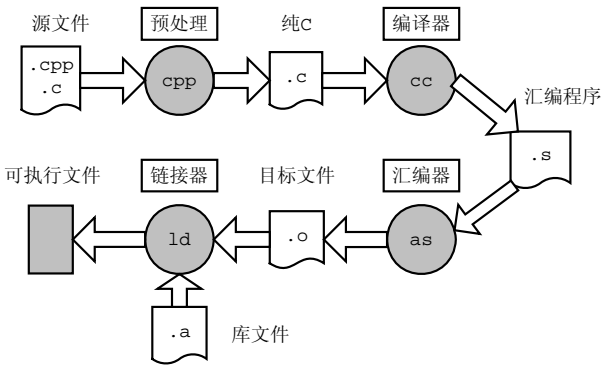


图 4-2 C 语言编译过程

12. 计算机语言说明

数学是描述大自然的语言，会计是人们记录商业活动的语言，人类语言（英语、汉语、法语、西班牙语等）则是人与人之间交流的语言，计算机语言是人们和计算机（CPU）交流的语言。

由于 CPU 只能理解机器语言（CPU 指令集），而人们对机器语言（一串二进制数）很难阅读和调试，于是就产生了汇编语言。汇编语言的特点是利用助记符代替相应的机器语言二进制指令，如利用“ADD”助记符代替二进制指令“1000001100000110”，汇编语言较机器语言更有利于人们编程，人们编写的汇编源程序再经汇编编译器翻译成机器语言程序（执行码），执行码就是 CPU 能理解的二进制指令。

由于汇编语言是面向机器的低级语言，与 CPU 的指令集息息相关，是按 CPU 的“思考”方式编写而成的计算机语言。汇编语言与人类思考方法相差较大，于是产生了面向过程的语言（PASCLE、C 等）和面向对象的语言（C++、Java 等）。但是计算机只能理解机器语言（CPU 指令集），人们编写的 C 语言、C++语言、Java 语言源程序要翻译成机器语言，CPU 才能够执行，执行翻译工作的是编译程序。就这样，一门新的计算机学科（编译原理）产生了，编译原理介绍了编译程序的实现方法。masm 是汇编的编译程序，gcc 是 Linux 下 C 语言和 C++语言的编译程序，JVC 是 Java 的编译程序。

万物皆流，万物归宗。计算机源程序，无论是高级语言 C、C++、Java 源程序，还是汇编语言源程序，最终都要翻译成机器语言，CPU 才能够执行。

13. CPU 的独白

嗨，大家好！我叫CPU，是中央处理器（Central Processing Unit）的简称，是电子计算机的主要设备之一，其功能主要是解释计算机指令及处理计算机软件中的数据，所谓的计算机的可编程性主要是指对CPU的编程。

在我看来，外部一切都是地址，我只负责从地址上取数据，然后计算数据，计算完毕向地址上存数据。我的工作主要是围绕着存数据、取数据和计算数据在进行。由于外界的地址只有内存和外设两种，CPU 引脚 M/\overline{IO} 高电平意味着与内存通信，低电平意味着与外设端口通信。地址以一个字节（8 位二进制数）为单位进行编排，微机中内存地址和外设端口地址是单独进行编排的，当我看到 MOV 指令就知道是与内存通信，看到 IN 和 OUT 指令就知道是与外设端口通信。

在我的眼里，地址上（内存和外设端口）的数据都是一堆二进制数，没有类型，没有任何含义。只有当我开始运行机器语言代码时，这些数据才变得有意义。我是一个优秀且卓越的执行者，完全按照机器语言代码功能进行执行。

人们通过布尔逻辑、数字电路技术把沙粒（二氧化硅）变成了我，我的世界只有 100 多个或几百多个机器指令，这些机器指令早已由 INTEL 等芯片厂商设计好。设计完成后，机器指令是固定的，不能再修改。因为我是硬件，不同于软件，设计生产完成后，要么是成功的芯片，要么是失败的芯片，没有第三条道路可走。

在我的身体里，利用寄存器暂存数据，利用运算器完成数据运算，利用控制器控制数据与内存或外设的传输。由于我工作的关系，我深刻理解什么叫分层，什么叫抽象，什么叫协议，什么

叫分工，什么叫转换，什么叫约定，什么叫专业，什么叫透明，什么叫物理，什么叫逻辑。其实计算机技术实现是哲学思想的体现，计算机技术较好的利用分层、抽象、模块化等思想使复杂的问题简单化。

人们常说“谢谢你的存在，世界因你而精彩”。然而我的世界是枯燥的，就是不断地正确执行这一百多个机器指令。也许专业就是简单的事情重复地做，我太过专业，人们赋予我计算专家头衔。

复杂的事情模块化，模块的事情简单化，简单的事情流程化，流程的事情自动化。计算机完成的功能的确非常复杂，要做好这项工作必须懂得分工和协作，每个人只做自己擅长的事情。所以我和我的伙伴们有明确的分工，由于责任明确，大家都完全遵守着各种协议和约定，所以我们配合得很默契，工作得很高效。我和我的同伴们是世界上最好的学生，从来规规矩矩，完全遵守各种协议和约定。内存完成工作数据的存储，硬盘完成长久数据的保存，键盘完成字符的输入，鼠标完成图形按钮的控制，显卡完成显示数据的转换，显示器完成图形显示，声卡完成数字声音向模拟声音的转换，音箱完成声音的播放。我和我的伙伴们总是呆在固定的地点，各自完成自己的工作，大家“鸡犬之声相闻，老死不相往来”，只是通过总线传递着数据进行交流。

许多人每天的工作和生活与我们联系在一起，人们与我们电脑从陌生到熟悉，然而许多人觉得我们太有内涵，难以弄懂我们复杂的工作原理和工作个性，又觉得从熟悉走向陌生。其实计算机的世界是按照一系列的协议和规则在运行，就像大自然按照一系列规则在运行一样。大自然是造物主创造的规则，而计算机则是全世界计算机专家们创造的协议与规则。了解计算机首先要了解各个硬件功能及硬件之间如何分工协作，然后再要知道软件的工作原理。软件是建立在硬件之上，就像精神建立物质之上一样。

许多人把我们电脑当做自我精神愉悦的朋友。许多人说想每天“听听你悦耳的声音，想看看你迷人的笑容，看见你我有一种莫名的快乐”。然而我的声音（声音格式）有各种格式，我的笑容（图像显示）有各种协议，我的声音和笑容凝聚着全球 IT 工程师辛勤的汗水和智慧的结晶，能为大家带来快乐我也感到很开心。我们电脑被人们制造出来后，然后我们改变了世界上人们的工作与生活。有时我们分不清是世界改变了我们，还是我们改变了世界，也许这是相互促进的结果。在我看来，电脑和人脑有相通之处，人脑的思维结构是由天生注定的，人可以通过学习和思考来优化思维软件。电脑的物理结构是由硬件决定的，而软件则是由 IT 工程师编写出来的，软件通过硬件来实现电脑价值的提升。

由于我 CPU 是一板一眼，只认识数字世界，人们觉得我 CPU 很难沟通。大家觉得我 CPU 是冰冷的芯下面藏着热情的火焰，在计算机发展初期，只有能编写计算机机器指令的人才能点燃 CPU 芯片中的火焰。由于机器指令晦涩难懂，这不适应电脑的发展与普及，于是编译器产生了，编译器是我 CPU 与计算机语言之间的翻译官。计算机语言是面向人类思维的语言，而我 CPU 只认识机器指令，所以编译器架起了我们沟通的桥梁。人们的许多工作都是效率、成本、功用三者的平衡，编译器让人们编写软件更加高效、成本更低、功能更强，编译器推动了软件业的繁荣。不管人们用多少种软件语言，编写软件复杂程度有多高，但最终都要翻译成我能理解的为数不多的上百个或几百个机器指令。我用这数量极少的指令展现了色彩缤纷的世界、千变万化的声音。这一切的结果是基于一定的规则和协议的，科学就是认识万事万物的规则和规律，世界上只有有规则和规律的事情才有意义，美妙的声音是有规律的，而噪声是无规律的。我深深懂得规则和规律的重要，

我的一切工作都是按照预定的规则和规律在进行。只有符合规则、约定、协议的数据，我和我的同伴们才能相互理解，理解是需要建立在协议、约定和规则的基础上。

在电脑世界里，标准的力量常常是无穷的，计算机业标准比任何其他行业都使用得广泛，顺标准则昌，逆标准则亡。由于全世界电脑各种协议需要统一的标准，才能更好地把世界软硬件厂商联系起来，推动计算机产业的进步，所以全世界许多 IT 公司都在认识标准、利用标准、制定标准、优化标准中博弈，如高清 DVD 标准之争。

计算机的世界，是数字（数据）的世界。计算机软硬件所做的一切只不过是為了数据的加工、转换、表现（显示、声音）、传输和存储。围绕数据在计算机中的处理产生了许多计算机学科，如数据结构是对数据的算法处理、数据库是对数据的关系处理、计算机网络是对数据的传输处理，而计算机存储是对数据的存储处理、计算机图形学是对数据的转换和显示处理。数据的加工、转换、表现、传输和存储都需要遵照一定的协议和规范。

4.2 常量与变量

C 语言中数据由常量和变量组成，常量顾名思义是其值不可变的量，变量顾名思义是其值可以改变的量。C 语言变量都有其数据类型，说明其在 C 语言中的类型。数据类型决定了数据占内存的字节数、数据的取值范围和其上可进行的操作。

1. 数据基本类型的分类及特点

表 4-1 列出了 C 语言数据基本类型及其说明，使用时要注意这些基本类型的数值范围。

表 4-1 C 语言数据基本类型分类表

	类型说明符	字 节	数值范围
字符型	char	1	C 字符集
基本整型	int	2	-32768~32767
短整型	short int	2	-32768~32767
长整型	long int	4	-214783648~214783647
无符号型	unsigned	2	0~65535
无符号长整型	unsigned long	4	0~4294967295
单精度实型	float	4	3/4E-38~3/4E+38
双精度实型	double	8	1/7E-308~1/7E+308

2. 标识符

标识符是用来标识变量名、符号常量名、函数名、数组名、类型名、文件名的有效字符序列。标识符只能由字母、数字、下画线组成，且第一个字母必须是字母或下画线；大小写敏感；不能使用关键字；长度最长 32 个字符；命名原则要见名知意，做到“顾名思义”。

3. 常量和符号常量

在程序执行过程中，其值不发生改变的量称为常量。常量有直接常量和符号常量两种。

① 直接常量

直接常量有整型常量、实型常量、字符常量、字符串常量、指针常量五种类型。下面是它们的举例说明。

整型常量：12、0、-3。

实型常量：4.6、-1.23。

字符常量：'a'、'b'。

② 符号常量：用标识符代表一个常量。

在C语言中，可以用一个标识符来表示一个常量，称之为符号常量。

符号常量在使用之前必须先定义，其一般形式为：

```
#define 标识符 常量
```

其中#define 是一条预处理命令（预处理命令都以“#”开头），称为宏定义命令，其功能是把该标识符定义为其后的常量值。

习惯上符号常量的标识符用大写字母，变量标识符用小写字母，以示区别。

【例 4-1】 符号常量的使用。

price.c 源代码如下：

```
#include <stdio.h>
#define PRICE 30
void main()
{
    int num,total;
    num=10;
    total=num* PRICE;
    printf("total=%d\n", total);
}
```

由上面实例看出，符号常量有以下特点和好处：

- ① 符号常量是用一个标识符代表一个常量。
- ② 符号常量与变量不同，它的值在其作用域内不能改变，也不能再被赋值。
- ③ 使用符号常量的好处是含义清楚，能做到“一改全改”。

4. 变量

（1）变量说明

其值可以改变的量称为变量，变量在内存中占据一定的存储单元。变量定义必须放在变量使用之前，一般放在函数体的开头部分，不能使用没有定义的变量。每个变量都有一个名字，称为变量名，其值存放在内存中，变量名是变量值的代号。

假设 `int a=0`，图 4-3 给出了此变量的内存布局。此变量的变量名为 `a`，其值为 0，程序执行时分配内存空间的首地址为 2012，即`&a` 等于 2012。

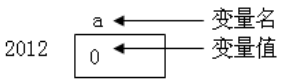


图 4-3 变量的内存布局

变量可以理解为内存相应位置所存数据的抽象，变量类型决定了变量所占空间的大小。字符型变量占用内存空间 1 个字节，整型变量占用内存空间 4 个字节。上述变量 `a` 可以理解为地址为 2012-2009 的 4 个字节内存空间的抽象，其存储的值为 0。一个 C 语言源程序编译成可执行程序后，对变量 `a` 的操作实际转换为对内存地址 2012-2009 的存取操作。

(2) 变量初始化

初始化是指在变量定义的同时，给变量赋以初值的方法。

变量定义中赋初值的一般形式如下：

类型说明符 变量 1= 值 1,变量 2= 值 2,……;

例如：

```
int a=3;
int b,c=5;
```

(3) 变量转换

变量转换分为显式转换和隐式转换两种。

其中，隐式转换分为运算转换、赋值转换、输出转换和函数调用转换。隐式转换时变量的转化顺序为 `char,short→int→unsigned→long→float→double`。

显式转换的一般形式为：(类型名)(表达式)，例如，`(float)a` 是把 `a` 转换为实型。

5. 整型数据

整型数据有前缀表示法和后缀表示法两种，具体说明如下。

① 前缀表示法。以 0 开头表示八进制，以 0x (或 0X) 表示十六进制，默认表示十进制。015 (十进制为 13)，0X2A (十进制为 42)。

② 后缀表示法。L (或 l) 表示长整型，U (或 u) 表示无符号数。

整型变量赋值要注意其取值范围，超过范围会造成变量的溢出。

6. 转义字符

转义字符是一种特殊的字符常量。转义字符以反斜线“\”开头，后跟一个或几个字符。转义字符具有特定的含义，不同于字符原有的意义，故称“转义”字符。例如，`printf` 函数格式串中用到的“\n”就是一个转义字符，其意义是“回车换行”。转义字符主要用来表示用一般字符不便于表示的控制代码。表 4-2 列出了常用的转义字符及其含义。

表 4-2 常用的转义字符及其含义

转义字符	转义字符的意义	ASCII 代码
\n	回车换行	10
\t	横向跳到下一制表位置	9
\b	退格	8
\r	回车	13
\f	走纸换页	12
\\	反斜线字符"\ "	92
\'	单引字符	39
\"	双引字符	34
\a	鸣铃	7
\ddd	1~3 位八进制数所代表的字符	
\xhh	1~2 位十六进制数所代表的字符	

广义地讲，c 语言字符集中的任何一个字符均可用转义字符来表示。表中的\ddd 和\xhh 正是为此而提出的，ddd 和 hh 分别为八进制和十六进制的 ASCII 代码。例如，\101 表示字母 A，\102 表示字母 B，\x0A 表示换行等。

7. 字符常量

字符常量是用单引号引起来的一个字符。

例如：'a'、'b'、'='、'+'、'?'都是合法字符常量。

在 c 语言中，字符常量有以下特点：

- ① 字符常量只能用单引号引起来，不能用双引号或其他括号。
- ② 字符常量只能是单个字符，不能是字符串。
- ③ 字符可以是字符集中任意字符，但数字被定义为字符型之后就不能参与数值运算。例如，'5'和 5 是不同的，'5'表示字符 5，对应 ASCII 码 35，而 5 对应 ASCII 码 5。

8. 字符变量

字符变量用来存储字符常量，即单个字符，字符变量的类型说明符是 char。

每个字符变量被分配一个字节的内存空间，字符值是以 ASCII 码的形式存放在变量的内存单元之中的。

c 语言允许对整型变量赋以字符值，也允许对字符变量赋以整型值。在输出时，允许把字符变量按整型量输出，也允许把整型量按字符量输出。

短整型量为二字节量，字符量为单字节量，当整型量按字符型量处理时，只有低八位字节参与处理。

例如，x 的十进制 ASCII 码是 120，对字符变量 a 赋予'x'可以是 a='x'或 a=120，这两

条语句是等价的。

字符变量赋值还可以使用转义字符，如 `c='\0'` 与 `c=0`，这两条语句也是等价的。

【例 4-2】 字符变量使用举例。

`charint.c` 源代码如下：

```
#include <stdio.h>
void main()
{
    char a,b;
    a=120;
    b='y';
    printf("%c,%c\n",a,b);
    printf("%d,%d\n",a,b);
}
```

编译 `gcc charint.c -o charint`。

执行 `./charint`，执行结果如下：

```
x,y
120,121
```

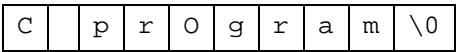
9. 字符串常量

字符串常量是由一对双引号引起来的字符序列，如 `"CHINA"`、`"C program"`、`"$12.5"` 等都是合法的字符串常量。

字符串常量和字符常量是不同的量。它们之间主要有以下区别：

- ① 字符常量由单引号引起来，字符串常量由双引号引起来。
- ② 字符常量只能是单个字符，字符串常量则可以包含一个或多个字符。
- ③ 字符常量占一个字节的内存空间。字符串常量占的内存字节数等于字符串中字节数加 1。增加的一个字节中存放字符 `"\0"`（ASCII 码为 0），这是字符串结束的标志。

例如，字符串 `"C program"` 在内存中所占的字节为：

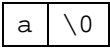


字符常量 `'a'` 和字符串常量 `"a"` 虽然都只有一个字符，但在内存中的情况是不同的，两者占用内存情况说明如下。

`'a'` 在内存中占一个字节，可表示为：



`"a"` 在内存中占两个字节，可表示为：



4.3 运算符

C 语言中运算符和表达式数量之多，在高级语言中是少见的。正是丰富的运算符和表达式使 C 语言功能十分完善，这也是 C 语言的主要特点之一。

C 言的运算符不仅具有不同的优先级，而且还有一个特点，就是它的结合性，使用时要注意运算符是右结合性还是左结合性的。

1. C 语言的运算符分类

对 C 语言的运算符分类说明如下：

- ① 算术运算符：用于各类数值运算，包括加 (+)、减 (-)、乘 (*)、除 (/)、求余（或称模运算%）、自增 (++)、自减 (--)，共七种。
- ② 关系运算符：用于比较运算，包括大于 (>)、小于 (<)、等于 (==)、大于等于 (>=)、小于等于 (<=) 和不等于是 (!=) 六种。
- ③ 逻辑运算符：用于逻辑运算，包括与 (&&)、或 (||)、非 (!) 三种。
- ④ 位操作运算符：参与运算的量，按二进制位进行运算，包括位与 (&)、位或 (|)、位非 (~)、位异或 (^)、左移 (<<)、右移 (>>) 六种。
- ⑤ 赋值运算符：用于赋值运算，分为简单赋值 (=)、复合算术赋值 (+=、-=、*=、/=、%=) 和复合位运算赋值 (&=、|=、^=、>>=、<<=)，三类共 11 种。
- ⑥ 条件运算符：这是一个三目运算符，用于条件求值 (?:)。
- ⑦ 逗号运算符：用于把若干表达式组合成一个表达式 (,)。
- ⑧ 指针运算符：用于取内容 (*) 和取地址 (&) 两种运算符。
- ⑨ 求字节数运算符：用于计算数据类型所占的字节数 (sizeof)。
- ⑩ 特殊运算符：有优先级 ()、下标 []、结构成员 (→、.)、取负运算符-、强制类型转换 (类型)。

2. 基本的算术运算符

对基本的算术运算符说明如下：

- ① 加法运算符 “+”：加法运算符为双目运算符，即应有两个量参与加法运算。如 a+b、4+8 等，加法运算符具有左结合性。
- ② 减法运算符 “-”：减法运算符为双目运算符。但 “-” 也可作为负值运算符使用，此时为单目运算，如 -x、-5 等具有左结合性。

- ③ 乘法运算符 “*”: 双目运算符，具有左结合性。
- ④ 除法运算符 “/”: 双目运算，具有左结合性。参与运算量均为整型时，结果也为整型，舍去小数。如果运算量中有一个是实型，则结果为双精度实型。
- ⑤ 求余运算符（模运算符）“%”: 双目运算符，具有左结合性。要求参与运算的量均为整型，求余运算的结果等于两数相除后的余数。

算术单目运算符为右结合性，双目运算符为左结合性，两整数相除，结果为整数，%要求两侧均为整型数据。

3. 表达式、运算符的优先级和结合性

表达式是由常量、变量、函数和运算符组合起来的式子。一个表达式有一个值及其类型，它们等于计算表达式所得结果的值和类型。表达式求值按运算符的优先级和结合性规定的顺序进行。单个的常量、变量、函数可以看做是表达式的特例。

算术表达式是由算术运算符和括号连接起来的式子。下面是对算术表达式和算术运算符的详细说明。

算术表达式：用算术运算符和括号将运算对象（也称操作数）连接起来的，符合 C 语法规则的式子。

运算符的结合性：C 语言中运算符的结合性分为两种，即左结合性（自左至右）和右结合性（自右至左），如算术运算符的结合性是自左至右，即先左后右，现有表达式 $x-y+z$ ，则 y 应先与“-”号结合，执行 $x-y$ 运算，然后再执行 $+z$ 的运算。这种自左至右的结合方向就称为“左结合性”，而自右至左的结合方向称为“右结合性”。最典型的右结合性运算符是赋值运算符，如 $x=y=z$ ，由于“=”的右结合性，应先执行 $y=z$ 再执行 $x=(y=z)$ 运算。C 语言运算符中有不少为右结合性，应注意区别，以避免理解错误。

运算符的优先级：C 语言中，运算符的运算优先级共分为 15 级，1 级最高，15 级最低。在表达式中，优先级较高的先于优先级较低的进行运算。而当在一个运算量两侧的运算符优先级相同时，则按运算符的结合性所规定的结合方向处理。

表 4-3 列出了 C 语言运算符优先级及其说明，此表了解即可，无须记忆，编程时优先级的确定时可使用圆括号“()”来解决。

表 4-3 C 语言运算符优先级表

优先级	运算符	名称或含义	使用形式	结合方向	说 明
1	()	圆括号	(表达式) / 函数名 (形参表)	左到右	
	[]	数组下标	数组名 [常量表达式]		
	.	成员选择 (对象)	对象 . 成员名		
	->	成员选择 (指针)	对象指针->成员名		
2	-	负号运算符	-表达式	右到左	单目运算符
	(类型)	强制类型转换	(数据类型) 表达式		

续表

优先级	运算符	名称或含义	使用形式	结合方向	说 明
2	++	自增运算符	++变量名/变量名++		单目运算符
	--	自减运算符	--变量名/变量名--		单目运算符
	*	取值运算符	*指针变量		单目运算符
	&	取地址运算符	&变量名		单目运算符
	!	逻辑非运算符	!表达式		单目运算符
	~	按位取反运算符	~表达式		单目运算符
	sizeof	长度运算符	sizeof (表达式)		
3	/	除	表达式/表达式	左到右	双目运算符
	*	乘	表达式*表达式		双目运算符
	%	余数（取模）	整型表达式/整型表达式		双目运算符
4	+	加	表达式+表达式	左到右	双目运算符
	-	减	表达式-表达式		双目运算符
5	<<	左移	变量<<表达式	左到右	双目运算符
	>>	右移	变量>>表达式		双目运算符
6	>	大于	表达式>表达式	左到右	双目运算符
	>=	大于等于	表达式>=表达式		双目运算符
	<	小于	表达式<表达式		双目运算符
	<=	小于等于	表达式<=表达式		双目运算符
7	==	等于	表达式==表达式	左到右	双目运算符
	!=	不等于	表达式!= 表达式		双目运算符
8	&	按位与	表达式&表达式	左到右	双目运算符
9	^	按位异或	表达式^表达式	左到右	双目运算符
10		按位或	表达式 表达式	左到右	双目运算符
11	&&	逻辑与	表达式&&表达式	左到右	双目运算符
12		逻辑或	表达式 表达式	左到右	双目运算符
13	?:	条件运算符	表达式 1? 表达式 2: 表达式 3	右到左	三目运算符
14	=	赋值运算符	变量=表达式	右到左	
	/=	除后赋值	变量/=表达式		
	=	乘后赋值	变量=表达式		
	%=	取模后赋值	变量%=表达式		
	+=	加后赋值	变量+=表达式		
	-=	减后赋值	变量-=表达式		
	<<=	左移后赋值	变量<<=表达式		
	>>=	右移后赋值	变量>>=表达式		
	&=	按位与后赋值	变量&=表达式		
	^=	按位异或后赋值	变量^=表达式		
	=	按位或后赋值	变量 =表达式		
15	,	逗号运算符	表达式, 表达式, ...	左到右	从左向右顺序运算

强制类型转换运算符

强制类型转换运算符的形式为：

(类型说明符) (表达式)

其功能是把表达式的运算结果强制转换成类型说明符所表示的类型。

例如：

```
(float)a      把 a 转换为实型
(int)(x+y)    把 x+y 的结果转换为整型
```

自增、自减运算符

自增 1：自增 1 运算符记为“++”，其功能是使变量的值自增 1。

自减 1：运算符记为“--”，其功能是使变量值自减 1。

自增 1、自减 1 运算符均为单目运算，都具有右结合性，可有以下几种形式：

++i: i 自增 1 后再参与其他运算

--i: i 自减 1 后再参与其他运算

i++: i 参与运算后，i 的值再自增 1

i--: i 参与运算后，i 的值再自减 1

【例 4-3】 自增、自减运算符举例。

varadd.c 源代码如下：

```
#include <stdio.h>
void main()
{
    int i=8;
    int a ;
    printf("i=%d\n",++i);
    printf("i=%d\n",i);
    a=++i ;
    printf("a=%d, i=%d\n", a, i);
    a=i++ ;
    printf("a=%d, i=%d\n", a, i);
    a--i ;
    printf("a=%d, i=%d\n", a, i);
    a=i-- ;
    printf("a=%d, i=%d\n", a, i);
    printf("i=%d\n",--i);
    printf("i=%d\n",i++);
    printf("i=%d\n",i--);
    printf("i=%d\n",-i++);
    printf("i=%d\n",-i--);
}
```

编译 `gcc varadd.c -o varadd`。

执行 `./varadd`，执行结果如下：

```
i=9
i=9
a=10, i=10
a=10, i=11
a=10, i=10
a=10, i=9
i=8
i=8
i=9
i=-8
i=-9
```

4. 赋值运算符和赋值表达式

(1) 赋值运算符

简单赋值运算符为“=”，由“=”连接的式子称为赋值表达式。其一般形式如下：

变量=表达式

例如：

```
x=a+b;
```

`a=b=c=5` 可理解为 `a=(b=(c=5))`。

(2) 类型转换

如果赋值运算符两边的数据类型不相同，系统将自动进行类型转换，即把赋值号右边的类型换成左边的类型。具体规定如下：

① 实型赋予整型，舍去小数部分。

② 整型赋予实型，数值不变，但将以浮点形式存放，即增加小数部分（小数部分的值为 0）。

③ 字符型赋予整型，由于字符型为一个字节，而整型为两个字节，故将字符的 ASCII 码值放到整型量的低八位中，高八位为 0。整型赋予字符型，只把低八位赋予字符量。

(3) 复合的赋值运算符

在赋值符“=”之前加上其他二目运算符可构成复合赋值运算符，如 `+=`、`-=`、`*=`、`/=`、`%=`、`<<=`、`>>=`、`&=`、`^=`、`|=` 都是复合赋值运算符。

构成复合赋值表达式的一般形式为：

变量 双目运算符=表达式

它等效于：

变量=变量 运算符 表达式

例如：

a+=5 等价于 a=a+5;

x*=y+7 等价于 x=x*(y+7);

复合赋值符这种写法，对初学者可能不习惯，但十分有利于编译处理，能提高编译效率并产生质量较高的目标代码。

5. 逗号运算符和逗号表达式

在 C 语言中逗号“,”也是一种运算符，称为逗号运算符，其功能是把多个表达式连接起来组成一个表达式，称为逗号表达式。

其一般形式为：

表达式 1,表达式 2,⋯,表达式 n

其求值过程是依次求表达式的值，并以表达式 n 的值作为整个逗号表达式的值。

6. 关系运算符

关系表达式的语法形式如下：

表达式 关系运算符 表达式

表 4-4 列出了 C 语言六种关系运算符。在 C 语言中，=表示赋值，==表示数学中的相等关系，使用时请注意=和==的区别。

表 4-4 关系运算符表

运 算 符	含 义
==	等于
!=	不等于
>	大于
<	小于
>=	大于或等于
<=	小于或等于

对于关系表达式，如果表达式所表示的比较关系成立则值为真 (True)，否则为假 (False)，在 C 语言中分别用 int 型的 1 和 0 表示真和假。如果变量 x 的值是 -1，那么 x>0 这个表达式的值为 0，x>-2 这个表达式的值为 1。

关系运算符的两个操作数应该是相同类型的，两边都是整型或者都是浮点型时可以做比较，但两个字符串不能做比较，需要用字符串函数进行比较。

7. 逻辑运算符

C 语言中提供了三种逻辑运算符：&&是与运算 (AND) 符，||是或运算 (OR) 符，!是非运

算（NOT）符。

逻辑表达式的语法形式如下：

表达式 逻辑运算符 表达式

(1) 逻辑运算符优先级

逻辑运算符优先级顺序为：！（非）>&&（与）>||（或）。

图 4-4 给出了运算符优先级图，其中赋值运算符优先级最低，！优先级最高。

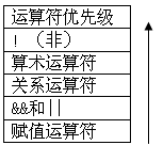


图 4-4 运算符优先级

按照运算符的优先级顺序可以得出：

a>b && c>d 等价于 (a>b)&&(c>d)

!b==c || d<a 等价于 ((!b)==c) || (d<a)

a+b>c&& x+y<b 等价于 ((a+b)>c)&&((x+y)<b)

(2) 逻辑运算的值

逻辑运算的值分为“真”和“假”两种，分别用“1”和“0”来表示。

与运算&&：参与运算的两个量都为真时，结果才为真，否则为假。例如，5>0 && 4>2 为真，即为 1。

或运算||：参与运算的两个量只要有一个为真，结果就为真，两个量都为假时，结果为假。例如，5>0 || 5>8 为真。

非运算!：参与运算量为真时，结果为假；参与运算量为假时，结果为真。例如，!(5>0)为假。

8. 条件运算符和条件表达式

如果在条件语句中，只执行单个的赋值语句时，常可使用条件表达式来实现。这样不但使程序简洁，也提高了运行效率。

条件运算符为?和:，它是一个三目运算符，即有三个参与运算的量。

由条件运算符组成条件表达式的语法形式如下：

表达式 1? 表达式 2: 表达式 3

其求值规则为：如果表达式 1 的值为真，则以表达式 2 的值作为条件表达式的值，否则以表达式 3 的值作为整个条件表达式的值。条件表达式通常用在赋值语句之中。

例如下面的条件语句：

```
if(a>b) max=a;
    else max=b;
```

可用条件表达式写为：

```
max=(a>b)?a:b;
```

该语句的语义为如果 $a>b$ 为真，则把 a 赋予 max ，否则把 b 赋予 max 。

使用条件表达式时，还应注意以下几点：

① 条件运算符的运算优先级低于关系运算符和算术运算符，但高于赋值符。

因此：

```
max=(a>b)?a:b
```

可以去掉括号而写为：

```
max=a>b?a:b
```

② 条件运算符 $?$ 和 $:$ 是一对运算符，不能分开单独使用。

③ 条件运算符的结合方向是自右至左。

例如， $a>b?a:c>d?c:d$ 应理解为 $a>b?a:(c>d?c:d)$

【例 4-4】 条件运算符使用举例。

expr.c 源代码如下：

```
#include <stdio.h>
void main()
{
    int a,b,max;
    printf("\n input two numbers:  ");
    scanf("%d%d",&a,&b);
    printf("max=%d",a>b?a:b);
}
```

编译 `gcc expr.c -o expr`。

执行 `./expr`，执行结果如下：

```
input two numbers:  30 90
max=90
```

4.4 C语言控制结构

C语言有三种基本流程结构，即顺序、选择、循环。选择结构通过 `if-else`、`case-switch`、`goto` 语句实现，循环结构通过 `while`、`for`、`do-while` 语句实现。

4.4.1 if 语句

1. if 语句三种形式

if 语句用于分支条件判断，其语法形式有如下三种。

(1) 第一种形式为基本形式，只有 if 关键字。

if 语句基本形式语法如下：

if(表达式) 语句

其语义是：如果表达式的值为真，则执行其后的语句，否则不执行该语句。其执行过程可表示为图 4-5。

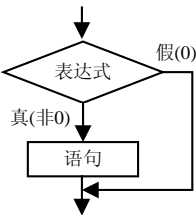


图 4-5 if 语句流程图

if 基本形式应用举例，if1.c 源代码如下：

```
#include <stdio.h>
void main()
{
    int a,b,max;
    printf("\n input two numbers:  ");
    scanf("%d%d",&a,&b);
    max=a;
    if (max<b) max=b;
    printf("max=%d",max);
}
```

编译 gcc if1.c -o if1。

执行 ./if1，执行结果如下：

```
input two numbers:    3 9
max=9
```

(2) 第二种形式为 if-else 形式。

if-else 语句语法形式如下：

```
if(表达式)
    语句 1;
else
    语句 2;
```

其语义是：如果表达式的值为真，则执行语句 1，否则执行语句 2。其执行过程可表示为

图 4-6。

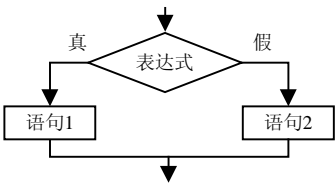


图 4-6 if-else 语句流程图

if-else 形式应用举例，if2.c 源代码如下：

```
#include <stdio.h>
void main()
{
    int a, b;
    printf("input two numbers:    ");
    scanf("%d%d",&a,&b);
    if(a>b)
        printf("max=%d\n",a);
    else
        printf("max=%d\n",b);
}
```

编译 gcc if2.c -o if2。

执行 ./if2，执行结果如下：

```
input two numbers:    10 0
max=10
```

(3) 第三种形式为 if-else-if 形式。

前两种形式的 if 语句一般都用于两个分支的情况。当有多个分支可供选择时，可采用 if-else-if 语句，其语法形式如下：

```
if(表达式 1)
    语句 1;
else if(表达式 2)
    语句 2;
else if(表达式 3)
    语句 3;
...
else if(表达式 m)
    语句 m;
else
    语句 n;
```

其语义是：依次判断表达式的值，当出现某个值为真时，则执行其对应的语句，然后跳到整个 if 语句之外继续执行程序。如果所有的表达式均为假，则执行语句 n，然后继续执行后续程序。其执行过程可表示为图 4-7。

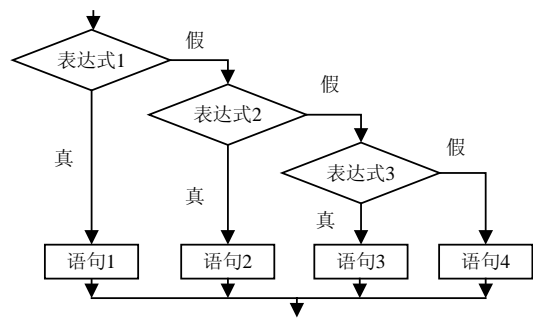


图 4-7 if-else-if 语句流程图

if-else-if 形式应用举例，if3.c 源代码如下：

```
#include <stdio.h>
void main()
{
    char c;
    printf("input a character:  ");
    c=getchar();
    if(c<32)
        printf("This is a control character\n");
    else if(c>='0'&&c<='9')
        printf("This is a digit\n");
    else if(c>='A'&&c<='Z')
        printf("This is a capital letter\n");
    else if(c>='a'&&c<='z')
        printf("This is a small letter\n");
    else
        printf("This is an other character\n");
}
```

编译 gcc if3.c -o if3。

执行 ./if3，执行结果如下：

```
input a character:  X
This is a capital letter
```

2. if 语句的嵌套

当 if 语句中的执行语句又是 if 语句时，则构成了 if 语句嵌套的情形，为了避免这种二义性，C 语言规定，else 总是与它前面最近的 if 配对。

下面的 if 语句就构成了 if 语句的嵌套，使用 if 语句应尽量避免此种形式，应尽量使用“{ }”来说明 else 匹配哪一个 if。

```
if(表达式 1)
    if(表达式 2)
        语句 1;
    else
        语句 2;
```

4.4.2 switch 语句

1. switch 语句说明

switch 语句用于多分支选择，其语法形式如下：

```
switch(表达式){
    case 常量表达式 1: 语句 1;
    case 常量表达式 2: 语句 2;
    ...
    case 常量表达式 n: 语句 n;
    default           : 语句 n+1;
}
```

其语义是：计算表达式的值，并逐个与其后的常量表达式值相比较，当表达式的值与某个常量表达式的值相等时，即执行其后的语句，然后不再进行判断，继续执行后面所有 case 后的语句。如表达式的值与所有 case 后的常量表达式均不相同，则执行 default 后的语句。所以 case 语句需要用 break 进行跳出，否则将对后面的语句顺序执行。

2. switch 语句注意事项

switch 有如下注意事项：

- ① 在 case 后的各常量表达式的值不能相同，否则会出现错误。
- ② 在 case 后，允许有多个语句，可以不用 {} 括起来。
- ③ 各 case 和 default 子句的先后顺序可以变动，且不会影响程序执行结果。
- ④ default 子句可以省略不用，但不提倡省略。
- ⑤ switch 表达式结果只能为整型变量和字符型变量。

3. switch 关键字应用举例

switch1.c 源代码如下：

```
#include <stdio.h>
void main()
{
    int a;
    printf("input integer number:  ");
    scanf("%d",&a);
    switch (a){
        case 1:printf("Monday\n");break;
        case 2:printf("Tuesday\n"); break;
        case 3:printf("Wednesday\n");break;
        case 4:printf("Thursday\n");break;
        case 5:printf("Friday\n");break;
        case 6:printf("Saturday\n");break;
        case 7:printf("Sunday\n");break;
        default:printf("error\n");
    }
}
```

编译 `gcc switch1.c -o switch1`。

执行 `./switch1`，执行结果如下：

```
input integer number: 1
Monday
```

`switch2.c` 源代码如下：

```
#include <stdio.h>
void main()
{
    float a,b;
    char c;
    printf("input expression: a+(-,*,/)b \n");
    scanf("%f%c%f",&a,&c,&b);
    switch(c){
        case '+': printf("%.2f\n",a+b);break;
        case '-': printf("%.2f\n",a-b);break;
        case '*': printf("%.2f\n",a*b);break;
        case '/': printf("%.2f\n",a/b);break;
        default: printf("input error\n");
    }
}
```

编译 `gcc switch2.c -o switch2`。

执行 `./switch2`，执行结果如下：

```
input expression: a+(-,*,/)b
3*7
21.00
```

4.4.3 goto 语句

1. goto 语句说明

`goto` 语句是一种无条件转移语句，其语法形式如下：

`goto` 语句标号；

标号是一个有效的标识符，这个标识符加上一个“:”一起出现在函数内某处，执行 `goto` 语句后，程序将跳转到该标号处并执行其后的语句。

标号必须与 `goto` 语句同处于一个函数中，但可以不在一个循环层中。

`goto` 语句经典使用场合为程序处理多个错误时，跳转到结尾进行错误处理，达到错误处理代码复用的目的，并减少 `return` 语句，其他场合不建议使用。

2. goto 语句应用举例

`goto.c` 源代码如下：

```
#include <stdio.h>
```

```
int main()
{
    int i=0;
    scanf("%d", &i) ;
    if ( i<0 )
    {
        goto err_end ;
    }
    if ( i >100 )
    {
        goto err_end ;
    }
    printf("input number validity:%d\n", i) ;
    return 0 ;
err_end:
    printf("input number overflow, please guess again\n");
    return -1 ;
}
```

编译 gcc goto.c -o goto。

执行 ./goto, 执行结果如下:

```
900
input number overflow, please guess again
```

4.4.4 while 语句

1. while 语句说明

while 语句用来实现循环结构，其语法形式如下：

```
while(表达式) 语句
```

其中，表达式是循环条件，语句为循环体。

while 语句的语义是：计算表达式的值，当值为真（非 0）时，执行循环体语句。其执行过程可用图 4-8 表示。

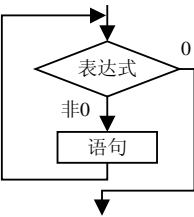


图 4-8 while 语句流程图

2. while 语句举例

用while语句求 $\sum_{n=1}^{100} n$ 。

while.c 源代码如下:

```
#include <stdio.h>
void main()
{
    int i,sum=0;
    i=1;
    while(i<=100)
    {
        sum=sum+i;
        i++;
    }
    printf("%d\n",sum);
}
```

编译 gcc while.c -o while。

执行 ./while, 执行结果如下:

```
sum=5050
```

4.4.5 do-while 语句

1. do-while 语句原型

do-while 语句同样是用来实现循环结构的, 其语法形式如下:

```
do
    语句
while(表达式);
```

这个循环与 while 循环的不同在于: 它先执行循环中的语句, 然后再判断表达式是否为真, 如果为真则继续循环; 如果为假, 则终止循环。因此, do-while 循环至少要执行一次循环语句。其执行过程可用图 4-9 表示。

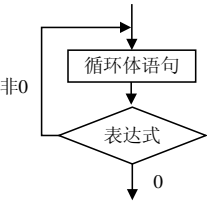


图 4-9 do-while 语句流程图

2. do-while 语句举例

用do-while语句求 $\sum_{n=1}^{100} n$ 。

dowhile.c 源代码如下:

```
#include <stdio.h>
void main()
{
```

```
int i,sum=0;
i=1;
do
{
    sum=sum+i;
    i++;
}while(i<=100)
printf("sum=%d\n",sum);
}
```

编译 gcc dowhile.c -o dowhile。

执行 ./dowhile, 执行结果如下:

```
sum=5050
```

4.4.6 for 语句

1. for 语句原型

for 语句也是用来实现循环结构的, 其语法形式如下:

```
for(表达式 1; 表达式 2; 表达式 3) 语句
```

它的执行过程如下:

- 1) 先求解表达式 1。
- 2) 求解表达式 2, 若其值为真 (非 0), 则执行 for 语句中指定的内嵌语句, 然后执行下面第 3 步; 若其值为假 (0), 则结束循环, 转到第 5 步。
- 3) 求解表达式 3。
- 4) 转回上面第 2 步继续执行。
- 5) 循环结束, 执行 for 语句下面的一个语句。

其执行过程可用图 4-10 表示。

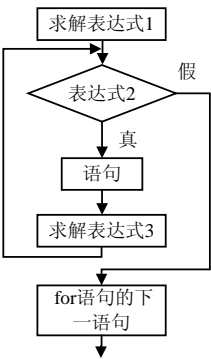


图 4-10 for 语句流程图

for 语句最简单的应用形式也是最容易理解的形式，其使用形式如下：

```
for(循环变量赋初值;循环条件;循环变量增量) 语句
```

注意 for 循环三个部分之间要用“;”分开，举例说明如下。

```
for(i=1; i<=100; i++)sum=sum+i; /*利用 for 语句实现上述 1 到 100 的相加功能*/
```

2. for 语句的多种使用方法

for 语句有如下多种使用方法。

```
for(i=1; i<=100; i++) /*常见使用方法*/
for(i=1;;i++)          /*死循环，语句中碰到合适条件时用 break 跳出*/
for(i=1;i<=100;)       /*循环增加放到语句中*/
for(;i<=100;)          /*初始化放到语句外，循环增量放到语句中*/
for(;;)                /*相当于 while(1)语句*/
```

3. for 语句应用举例

for.c 源代码如下：

```
#include <stdio.h>
void main()
{
    int i, j, k;
    printf("i j k\n");
    for (i=0; i<2; i++)
        for(j=0; j<2; j++)
            for(k=0; k<2; k++)
                printf("%d %d %d\n", i, j, k);
}
```

编译 gcc for.c -o for。

执行 ./for，执行结果如下：

```
i j k
i=0 j=0 k=0
i=0 j=0 k=1
i=0 j=1 k=0
i=0 j=1 k=1
i=1 j=0 k=0
i=1 j=0 k=1
i=1 j=1 k=0
i=1 j=1 k=1
```

4. 几种循环的比较

while、do-while、for 三种循环都可以用来处理同一个问题，一般可以互相代替，for 语句功能最强。

while 和 do-while 循环，循环体中应包括使循环趋于结束的语句。

用 while 和 do-while 循环时，循环变量初始化的操作应在 while 和 do-while 语句之前

完成，而 for 语句可以在表达式 1 中实现循环变量的初始化。

4.4.7 break 和 continue 语句

1. 语句说明

continue 语句终止当前循环后，又回到循环体的开头准备执行下一次循环。

break 语句跳出当前循环。

2. break 和 continue 语句举例

breakcon.c 源代码如下：

```
#include <stdio.h>
int is_prime(int n)
{
    int i;
    for (i = 2; i < n; i++)
        if (n % i == 0)
            break;
    if (i == n)
        return 1;
    else
        return 0;
}
int main(void)
{
    int i;
    for (i = 1; i <= 100; i++) {
        if (!is_prime(i))
            continue;
        printf("%d ", i);
    }
    printf("\n") ;
    return 0;
}
```

编译 gcc breakcon.c -o breakcon。

执行 ./breakcon，执行结果如下：

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79
83 89 97
```

程序说明如下：

① is_prime 函数从 2 到 n-1 依次检查有没有能被 n 整除的数，如果有，就说明 n 不是素数，立刻跳出循环而不执行 i++，所以 is_prime 函数中使用的是 break。

② 在主程序中，从 1 到 100 依次检查每个数是不是素数，如果不是素数，并不直接跳出循环，而是 i++后继续执行下一次循环，因此用 continue 语句。

第 5 章

C语言函数

5.1 函数简述

在学生时代的数学课上，老师用 $y=f(x,a,\dots)$ 来说明数学中的函数。C 语言是函数式语言，C 语言函数的名称其实也借鉴了数学中的函数。

函数是按照模块化设计思想，实现特殊控制流程的程序块。

函数在内存表现为内存中的一段二进制代码，可以被 CPU 执行的一段机器码。而程序的执行只不过是程序代码段的顺序执行和跳转执行而已。函数调用属于跳转执行代码，将程序代码段指针跳到函数起始地址处开始执行代码，函数执行完成后代码段指针指向调用函数程序的下一行。

1. 函数概述

函数的基本思想为将一个大的程序按功能分割成一些小模块。

函数特点为各模块相对独立、功能单一、结构清晰、接口简单，控制了程序设计的复杂性，提高元件的可靠性，缩短开发周期，避免程序开发的重复劳动，易于维护和功能扩充。函数开发方法为自上向下、逐步分解、分而治之。

C 是函数式语言，必须有且只能有一个名为 `main` 的主函数，C 程序的执行总是从 `main` 函数开始，在 `main` 中结束，函数不能嵌套定义，可以嵌套调用。

函数分类从用户角度可分为标准函数（库函数）和用户自定义函数。从函数调用或被调用形式上可分为有参函数和无参函数。从返回值上可划分为有返回值函数和无返回值函数，无返回值函数需要在函数名前加 `void` 关键字。

有参函数在函数定义及函数说明时都有参数，这些参数称为形式参数（简称为形参）。在函数调用时也必须给出参数，这些参数称为实际参数（简称为实参）。进行函数调用时，主调函数将把实参的值传送给形参，供被调函数使用。

使用库函数时应注意函数功能、函数参数的数目和顺序、各参数的意义和类型、函数返回值意义和类型、需要包含的头文件。

函数的返回值形式为：`return`（表达式）、`return` 表达式、`return` 三种。返回的作用是使程序控制从被调用函数返回到调用函数中，同时把返回值带给调用函数。函数中可有多个 `return` 语句（但良好的设计应保持只有一个 `return` 语句），若无 `return` 语句，遇 `}` 时，自动返回调用函数，若函数类型与 `return` 语句中表达式值的类型不一致，按照函数类型为准进行自动转换。函数返回值类型默认为 `int` 型。

函数调用形式为“函数名（实参表）”，调用时要求函数的实参与形参个数相等、类型一致、按顺序一一对应。

在同一文件，使用后面的函数（除返回值是 `int` 类型和 `void` 类型而且不带形参的函数）需要在文件头进行声明。如果有多个源文件，也可以把函数声明放在一个单独的头文件中，其他源文件包含此头文件。

2. 从用户角度划分函数

从用户角度可划分为库函数和用户自定义函数，具体说明如下：

① 库函数：由 C 系统提供，用户无须定义，也不必在程序中做类型说明，只须在程序前包含有该函数原型的头文件即可在程序中直接调用，如 `printf`、`scanf`、`strcpy`、`strcat` 等函数均属此类。

② 用户自定义函数：由用户按需要自己写的函数。对于用户自定义函数，不仅要在程序中定义函数本身，而且在主调函数程序中还必须对该被调函数进行类型说明，然后才能使用。

3. 从功能角度划分函数

从功能角度可以把 C 语言函数划分为如下 13 类：

① 字符类型测试函数：用于对字符类型进行测试。

② 转换函数：用于字符或字符串的转换，在字符量和各类数字量（整型、实型等）之间进行转换。

③ 目录路径函数：用于文件目录和路径操作。

④ 诊断函数：用于内部错误检测。

⑤ 图形函数：用于屏幕管理和各种图形功能。

⑥ 输入输出函数：用于完成输入输出功能。

⑦ 字符串函数：用于字符串操作和处理。

⑧ 内存管理函数：用于内存管理。

- ⑨ 数学函数：用于数学函数计算。
- ⑩ 日期和时间函数：用于日期、时间转换操作。
- ⑪ 进程控制函数：用于进程管理和控制。
- ⑫ 文件操作函数：用于对文件的操作。
- ⑬ 其他函数：用于其他各种功能的操作。

4. 程序执行时的内存布局

当一个源代码通过 gcc 编译成 a.out，执行 a.out 时，程序便开始了执行之旅（即进程）。操作系统为进程分配堆栈空间，随后把程序执行码放入文本段，把程序中经过初始化的全局变量和静态变量放入数据段，把程序中未初始化的全局变量和静态变量放入 bss 段，并对 bss 段数据初始化为 0。之后 CPU 代码段指针指向 main 的入口，CPU 堆栈段指针指向栈顶，代码段指针从 main 的入口地址顺序读取指令代码并进行执行，碰到局部变量和函数调用时，需要在栈顶分配空间并把堆栈段指针下移，碰到 malloc 等动态分配内存函数就在堆上分配内存。图 5-1 给出了程序执行时堆栈空间图，此图对于理解程序的运行机理非常重要。

栈段 (.stack) (局部于函数的数据)
堆段 (.heap) (malloc 申请的内存区域)
bss 段 (.bss) (未初始化的数据)
数据段 (.data) (经过初始化的全局变量和静态变量)
文本段 (.text) (又称代码段，存放程序执行码)

图 5-1 程序执行堆栈图

下面是对图 5-1 各段的具体说明。

.text：文本段，又称代码段，存放程序执行码。

.data：数据段，存放已初始化全局/静态变量，在整个程序执行过程中有效。

.bss：bss 段，存放未初始化全局/静态变量，在整个程序执行过程中有效。

.stack：栈段，存放函数调用栈和函数局部变量，其中的内容在函数执行期间有效，并由编译器负责分配和收回。

.heap：堆段，由程序显式分配和收回，如果不回收就会产生内存泄漏。

数据段和 bss 段有时也统称为数据区。

5.2 函数变量

1. 函数变量简述

在 C 语言中，每个变量和函数都有两个属性：数据类型和数据的存储类别。

变量存储类型的属性按生存期（时间）分为静态变量与动态变量，按照作用域（空间）分为局部变量与全局变量。

局部变量即内部变量，在函数内定义，只在本函数内有效。main 中定义的变量只在 main 中有效，所以也属于局部变量。不同函数中的同名变量，占用不同内存单元。函数中的形参属于局部变量。局部变量可用存储类型有 auto、register、static 三种类型，局部变量默认为 auto 类型。

静态存储是程序运行前分配固定存储空间，如 bss 段和数据段。

动态存储是程序运行期间根据需要动态分配的存储空间，如程序运行时使用的栈段和使用 malloc 函数申请空间使用的堆段。

静态变量从程序开始执行前分配空间到程序执行结束才释放空间。静态变量属于静态存储，静态变量包括全局变量、静态全局变量、静态局部变量。

动态变量作用域是从包含该变量定义的函数开始执行至此函数执行结束。动态变量属于动态存储，函数中的变量属于动态变量。

2. 变量存储类型关键字说明

变量的存储类型有 auto（自动型）、register（寄存器型）、static（静态型）、extern（外部型）四种，下面是这四种存储类型的具体说明。

① auto：局部变量。编译器在默认的情况下，所有变量都是 auto。

② register：寄存器变量。变量存在于 CPU 寄存器中，CPU 寄存器不足时此变量当做 auto 变量来处理。寄存器变量只能是单个值，长度小于或等于整型变量，由于此变量存在于 CPU 的寄存器中，不能用“&”来获得地址。把经常使用的变量放入寄存器中是为了获得高速度。

③ static：静态变量。在文件头定义的静态变量是全局静态变量，在函数中定义的静态变量是局部静态变量。静态变量在执行前将分配内存空间，在程序执行成后才释放内存空间。全局静态变量作用域为整个文件，局部静态变量作用域为单个函数。

④ extern：外部变量。引用的变量是其在其他文件头中进行定义的。

5.3 函数定义与调用

5.3.1 函数定义

函数使用前必须先定义，函数定义从函数调用或被调用形式上可分为有参函数和无参函数两

种，下面是对这两种函数定义形式的具体说明。

1. 无参函数的定义形式

无参函数定义一般形式如下：

```
类型标识符 函数名()  
{  
    变量定义部分  
    语句  
}
```

函数名是由用户定义的标识符，下面是一个无参也无返回值的函数。

```
void Hello()  
{  
    printf ("Hello,world \n");  
}
```

2. 有参函数定义的一般形式

有参函数比无参函数多了一个内容，即形式参数表列。在形参表中给出的参数称为形式参数，它们可以是各种类型的变量，各参数之间用逗号间隔。在进行函数调用时，主调函数将赋予这些形式参数实际的值。形参既然是变量，必须在形参表中给出形参的类型说明。

有参函数定义一般形式如下：

```
类型标识符 函数名(形参类型 [形参名],...)  
{  
    变量定义部分  
    语句  
}
```

例如，定义一个函数，用于求两个数中的大数，可写为：

```
int max(int a, int b)  
{  
    if (a>b) return a;  
    else return b;  
}
```

第一行说明 max 函数是一个整型函数，其返回的函数值是一个整数，形参 a 和 b 均为整型量。

3. 静态函数

用 static 修饰的函数为静态函数。静态函数的作用域范围局限于本文件，又称为内部函数。使用内部函数的好处是不同的人编写函数时，不用担心自己定义的函数，是否与其他文件中的函数同名。

5.3.2 函数的参数与返回值

1. 函数的形参与实参

发生函数调用时，主调函数把实参的值传送给被调函数的形参，从而实现主调函数向被调函

数的数据传送。

函数的形参和实参具有以下特点：

① C 语言函数调用方式为传值调用方式。

② C 语言函数调用时，为形参分配单元，并将实参的值复制到形参中。函数调用结束，形参单元被释放，实参单元仍保留并维持原值。

③ C 语言值传递的特点为形参与实参占用不同的内存单元，值为单向传递。

实参和形参在数量上、类型上、顺序上应严格一致，否则会发生类型不匹配的错误。

2. 函数的返回值

函数的值是指函数被调用之后，执行函数体中的程序段所取得的并返回给主调函数的值。

函数的值只能通过 return 语句返回主调函数。

return 语句的一般形式如下：

return 表达式；

或者为 return（表达式）；

或者为 return；

该语句的功能是计算表达式的值，并返回给主调函数。在函数中允许有多个 return 语句，但每次调用只能有一个 return 语句被执行，因此只能返回一个值。

函数返回值类型应与函数类型保持一致，如果两者不一致，则以函数类型为准，自动进行类型转换。

如果函数返回值为整型，在函数定义时可以省去类型说明。

没有返回值的函数，可以明确定义为“空类型”，类型说明符为“void”。

3. 形参实参、变量与返回值综合举例

（1）有参函数举例。

formreal.c 源代码如下：

```
#include <stdio.h>
int lab1 ;
int lab2 = 0 ;
static int lab3 ;
int formreal(int x, int y)
{
    static int lab4 ;
    int z ;
    printf("&lab4=%d\n", &lab4) ;
```

```
printf("&x=%d, &y=%d\n",&x, &y) ;
x= x*5 ;
y= y*5 ;
printf("x=%d, y=%d,lable1=%d,lable2=%d,lable3=%d,\
lable4=%d\n",x,y,lable1,lable2,lable3,lable4) ;
return 0 ;
}
int main()
{
    int a,b,c,d;
    printf("&lable1=%d,&lable2=%d,&lable3=%d\n", \
        &lable1, &lable2, &lable3 ) ;
    printf("&a=%d,&b=%d,&c=%d,&d=%d\n",&a,&b,&c,&d) ;
    printf("input two numbers:\n");
    scanf("%d%d",&a,&b);
    c=formreal(a,b);
    printf("a=%d, b=%d,c=%d, d=%d\n",a,b,c,d);
    return 0 ;
}
```

编译 gcc formreal.c -o formreal。

执行 ./formreal，执行结果如下：

```
&lable1=134518780,&lable2=134518768,&lable3=134518776
&a=-1075307972,&b=-1075307976,&c=-1075307980,&d=-1075307984
input two numbers:
2 3
&lable4=134518772
&x=-1075308016, &y=-1075308012
x=10, y=15,lable1=0,lable2=0,lable3=0,lable4=0
a=2, b=3,c=0, d=-1208725516
```

在上例中，x、y 属于形参，调用 c=formreal(a,b) 语句时，CPU 执行将堆栈指针下移，为函数、形参 x 与 y 分配内存单元，同时将实参 a 的值复制给形参 x，将实参 b 的值复制给形参 y。图 5-2 给出了函数调用时实参传值给形参的方式。

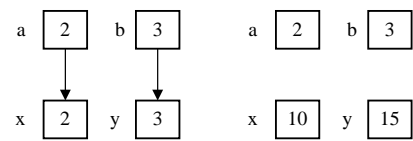


图 5-2 函数调用时实参传值给形参方式图

(2) formreal 程序执行堆栈表

表 5-1 列出了 formreal 程序执行时堆栈空间的内存模型，与实际执行堆栈空间并不完全一致。但可以让读者更好地理解全局变量、静态变量、实参和形参，特别是实参和形参是怎样进行传值调用的。

对表 5-1 设计的具体说明如下：

由于上面实例中使用的都是整型变量，整型变量占用内存空间为 4 个字节，所以这里以 4 个字节为单位对堆栈空间进行说明。

函数的返回值一般是通过 CPU 数据寄存器 EAX 返回，这里为了好理解，依然为函数返回值分配内存空间。

当执行 ./formreal 时，操作系统会为执行码分配好代码段、静态数据段、bss 段和栈段，然后代码指针指向 main 的入口。所以上述 lab1~lab4 变量是在程序执行前分配到数据段和 bss 段并完成赋值的。在这里要补充说明的是，静态局部变量作用范围是在编译时进行检查的，在编译后，静态局部变量和静态全局变量在执行程序看来，没有本质的差别。

从下面的执行过程可以看出，调用函数时系统为形参分配空间，将实参值复制给形参，所以说 C 语言传值调用为单向传值调用。

在代码编译成二进制码时，所有的变量失去意义，变量的操作其实都转化为对应内存地址的操作，变量其实只是一段内存空间值的抽象。编译完成后，机器代码中没有变量的存在，只有对内存线性逻辑地址的操作。

表 5-1 formreal 程序堆栈空间表

数据段说明	内存地址	实际内存
栈段	30000	main 函数返回值的存放空间
	29996	a
	29992	b
	29988	c
	29984	d
	29980	formreal 函数返回值的存放空间
	29976	x
	29972	y
	29968	z
	
堆段		
	
bss 段	10012	lab1
静态数据段	10008	lab3
	10004	lab4
	10000	lab2
代码段		1. formreal 函数地址入口 2. 在栈顶分配 z 变量空间 3. 打印 lab4 变量的地址到屏幕 4. 打印 x、y 变量的地址到屏幕 5. x 的值送到 CPU 乘以 5，数据结果由 CPU 返传到 x 的内存地址处 6. y 的值送到 CPU 乘以 5，数据结果由 CPU 返传到 y 的内存地址处 7. 打印 x、y、lab1~lab4 的值到屏幕上 8. 返回 0 给函数返回值空间，即 29980~29977 处 4 个字节的空间 9. 函数结束返回

续表

数据段说明	内存地址	实际内存
代码段		<div>10. main 函数地址入口</div> <div>11. 在栈顶分配 main 函数返回值、a、b、c、d 变量空间</div> <div>12. 打印 label1~label3 的地址到屏幕上</div> <div>13. 打印输入两个数字的提示信息</div> <div>14. 从屏幕上读取两个数字分别存放到 a、b 的内存空间里。此时&a 代表内存地址 29996，而 a 代表地址从 29993~29996 的 4 个字节内存抽象，即向这 4 个字节里存放第一个读取的值。b 变量同理</div> <div>15. 调用 formreal(a,b)函数，在栈顶分配 formreal 返回值空间、x 与 y 变量空间，并将实参 a 的值复制给 x，实参 b 的值复制给 y。然后即跳转到 l(formreal 函数入口)的地方去执行</div> <div>16. 将 formreal 函数的返回值赋给 c，即将 29980~29977 地址空间的值复制到 29988~29985。栈顶运行指针移到 29980 处，代表 formreal 函数运行栈空间已经释放</div> <div>17. 打印 a、b、c、d 的值到屏幕上</div> <div>18. 把返回值 0 赋值到 main 入口空间，即地址为 30000~29997 的内存空间处</div> <div>19. 程序执行完成，释放所有内存空间</div>

5.3.3 函数调用

1. 函数调用简述

函数声明是说明函数的调用形式，函数声明必须是已存在的函数。使用库函数须要包含对应的头文件，包含方法为#include <*.h>。使用用户自定义函数需在文件内进行函数声明或将函数声明放在自定义头文件中然后进行包含，包含用户自定义头文件方法为#include "*.h"。

函数声明的一般形式为：“函数类型 函数名(形参类型 [形参名],...)”或“函数类型 函数名()”。函数声明的作用是告诉编译系统函数类型、参数个数及类型，以便检验。函数定义与函数声明不同，函数定义是函数的实现，函数声明是说明函数的调用形式，方便其他程序按接口调用。函数声明位置可在函数内或外。

函数调用时实参必须有确定的值，形参必须指定类型，形参与实参类型一致，个数相同。若形参与实参类型不一致，自动按形参类型转换。形参在函数被调用前不占内存，函数调用时为形参分配内存，调用结束，内存释放。

函数定义不可嵌套，但可以嵌套调用函数。

2. 函数调用的方式

函数调用有如下三种方式：

① 函数表达式：函数作为表达式中的一项出现在表达式中，以函数返回值参与表达式的运算，这种方式要求函数是有返回值的。例如，z=max(x,y)是一个赋值表达式，把max的返回值赋予变量z。

② 函数语句：函数调用的一般形式加上分号即构成函数语句。例如，`printf ("%d",a)`和 `scanf ("%d",&b)`都是以函数语句的方式调用函数。

③ 函数实参：函数作为另一个函数调用的实际参数出现。这种情况是把该函数的返回值作为实参进行传送，因此要求该函数必须是有返回值的。例如，`printf("%d",max(x,y))`，即把 `max` 调用的返回值又作为 `printf` 函数实参来使用。在函数调用中还应该注意的一个问题是求值顺序的问题，所谓求值顺序是指对实参表中各量是自左至右使用还是自右至左使用，对此，各系统的规定不一定相同。

3. 被调用函数的声明

在主调函数中调用某函数之前应对该被调函数进行声明，这与使用变量之前要先进行变量说明是一样的。在主调函数中对被调函数做说明的目的是使编译系统知道被调函数返回值的类型，以便在主调函数中按此种类型对返回值做相应的处理。

其一般形式为：

类型说明符 被调函数名(类型 形参,类型 形参……) ；

或为：

类型说明符 被调函数名(类型,类型……) ；

函数声明时圆括号内需给出形参类型和形参名，或只给出形参类型，这便于编译系统进行检错，以防止可能出现的错误。

例如，`main` 函数中对 `max` 函数的说明如下：

```
int max(int a,int b);
```

或写为：

```
int max(int,int) ；
```

C 语言中规定在以下几种情况时，可以省去函数声明。

① 如果被调函数的返回值是 `int` 或 `void` 类型且没有形参时，可以不对被调函数做说明，而直接调用，这时系统将自动对被调函数返回值按整型处理。但不提倡不进行声明。

② 当被调函数的函数定义出现在主调函数之前时，在主调函数中也可以不对被调函数再做说明而直接调用。

③ 如在所有函数定义之前或头文件中对函数原型进行了声明，则在以后的各主调函数中，可不再对被调函数做说明。

4. 函数的递归调用

一个函数在它的函数体内调用它自身称为递归调用，这种函数称为递归函数。C 语言允许函

数的递归调用，在递归调用中，主调函数又是被调函数。执行递归函数将反复调用其自身，每调用一次就进入新的一层。直到碰到结束条件然后递归返回。递归的次数是有限的，常用的办法是加条件判断，满足某种条件后就不再做递归调用，然后逐层返回。

5. 用递归法计算 $n!$

(1) $n!$ 的递归程序实现

用递归法计算 $n!$ ，可用下述公式表示：

$$\begin{cases} n!=1 & (n=0, 1) \\ n \times (n-1)! & (n>1) \end{cases}$$

按公式编程实现如下，recursion.c 源代码如下：

```
#include <stdio.h>
int ff(int n)
{
    int f;
    if(n<0) printf("n<0,input error");
    else if(n==0||n==1) f=1;
    else f=ff(n-1)*n;
    return(f);
}
void main()
{
    int n;
    int y;
    printf("\n input a inteager number:\n");
    scanf("%d",&n);
    y=ff(n);
    printf("%d!=%ld",n,y);
}
```

编译 gcc recursion.c -o recursion。

执行 ./recursion，执行结果如下：

```
input a inteager number:
3
3!=6
```

程序中给出的函数 ff 是一个递归函数。主函数调用 ff 后即进入函数 ff 执行，如果 $n<0$ 、 $n==0$ 或 $n==1$ 时，都将结束函数的执行，否则就递归调用 ff 函数自身。由于每次递归调用的实参为 $n-1$ ，即把 $n-1$ 的值赋予形参 n ，最后当 $n-1$ 的值为 1 时形参 n 的值也为 1，将使递归终止，然后可逐层回退。

(2) recursion 程序执行堆栈表

表 5-2 列出程序 recursion 递归调用时堆栈变化情况。

表 5-2 递归调用堆栈表

数据段说明	内存地址	实际内存
栈段	30000	main 函数返回值存放空间
	29996	n =3 (main 中局部变量)
	29992	y
	29988	ff(3)
	29984	n=3 (ff 函数中局部变量, 下同, 不再说明)
	29980	f=ff(2)*3
	29976	ff(2)
	29972	n=2
	29968	f=ff(1)*2
	29964	ff(1)
	29960	n=1
	29956	f=1
	

↑
递归到 ff(1) 开始向上返回

根据表 5-2，对递归调用流程说明如下：

- ① 递归调用时，栈不停向下增长，直到碰到结束条件才依次向上返回。
- ② 如 ff(1)碰到结束条件 f=1 返回，此时 ff(1)=1。
- ③ 上一级 ff(2)中函数 f=ff(1)*2，f=1*2=2，f 返回给 ff(2)，ff(2)=2。
- ④ 再往上一级 ff(3)中 f=ff(2)*3，f=2*3=6，f 返回给 ff(3)，ff(3)=6。
- ⑤ 将 ff(3)赋值给 main 函数中变量 y，所以打印出来的值为 6。

第 6 章

C语言数组、结构体及指针

本章节介绍 C 语言数组、结构体及指针的知识，由于 C 语言中数组、结构体经常以指针的方式进行使用，所以将这三部分放入一个章节中。

6.1 C 语言数组

6.1.1 数组概述

在程序设计中，为了处理方便，把具有相同类型的若干变量按有序的形式组织起来，这些按序排列的同类数据元素的集合称为数组。在 C 语言中，数组属于构造数据类型。一个数组可以分解为多个数组元素，这些数组元素可以是基本数据类型或是构造类型。C 语言数组按维数分有一维、二维和 multidimensional，按数组元素的类型又可分为数值数组、字符数组、指针数组、结构体数组等各种类别。

对数组特点概括如下：

- ① 数组是有序数据的集合，用数组名标识。
- ② 数组元素须属同一数据类型，用数组名和下标确定。
- ③ 一维数组的定义：数据类型 数组名[常量表达式]。
- ④ 数组名表示的是内存首地址，是地址常量，所以只能是右值（=号的右边），不能是左值（=号的左边，当变量使用）。
- ⑤ 程序运行时对数组分配连续的内存空间，数组空间大小=数组维数*`sizeof`(元素数据类型)。
- ⑥ 数组元素个数的下标从 0 开始。
- ⑦ C 语言对数组不做越界检查，使用时要注意。例如，`int a[5]`只能用 `a[0]~a[4]`，用 `a[5]`就发生了越界。

- ⑧ 只能逐个引用数组元素，不能一次引用整个数组。
- ⑨ 数组不初始化时，其元素值为随机数。
- ⑩ 对 `static` 数组元素不赋初值时，系统会自动赋以 0 值。
- ⑪ 当为全部数组元素赋初值时，可不指定数组长度。
- ⑫ C 语言中无字符串变量，是用字符数组处理字符串，字符串结束标志为字符 `'\0'`。

6.1.2 一维数组

1. 一维数组定义

在 C 语言中使用数组必须先进行定义。

一维数组的定义方法如下：

类型说明符 数组名[常量表达式]；

其中：

- ① 类型说明符是任一种基本数据类型或构造数据类型。
- ② 数组名是用户定义的数组标识符。
- ③ 方括号中的常量表达式表示数据元素的个数，也称为数组的长度。

例如：

```
int a[10];           说明整型数组 a，有 10 个元素。  
  
float b[10],c[20];  说明实型数组 b，有 10 个元素；实型数组 c，有 20 个元素。  
  
char ch[20];        说明字符数组 ch，有 20 个元素。
```

对于数组类型的说明，应注意以下几点：

- ① 数组的类型实际是指数组元素的取值类型，对于同一个数组，其所有元素的数据类型都是相同的。
- ② 数组名的书写规则应符合标识符的书写规定。
- ③ 数组名不能与其他变量名相同。
- ④ 方括号中常量表达式表示数组元素的个数，但是其下标从 0 开始计算，如 `a[3]` 表示数组 `a` 有 3 个元素，这 3 个元素分别为 `a[0]`、`a[1]`、`a[2]`。
- ⑤ 注意数组的越界，达到数组的最大序号就是越界。
- ⑥ 不能在方括号中用变量来表示元素的个数，但是可用符号常数或常量表达式。

例如：

```
#define FD 5
main()
{
    int a[3+2],b[7+FD];
    .....
}
```

是合法的。

但是下述说明方式是错误的。

```
main()
{
    int n=5;
    int a[n];
    .....
}
```

2. 一维数组元素的引用

数组元素是组成数组的基本单元，数组元素也是一种变量，其标识方法为数组名后跟一个下标，下标表示了元素在数组中的顺序号。

数组元素引用形式如下：

数组名[下标]

其中，下标只能为整型常量或整型表达式，如为小数时，c 编译将自动取整。

例如，a[5]、a[i+j]、a[i++]都是合法的数组元素。

数组元素通常也称为下标变量，必须先定义数组，才能使用下标变量。在 c 语言中只能逐个地使用下标变量，而不能一次引用整个数组。

例如，输出有 10 个元素的数组必须使用循环语句逐个输出各下标变量，输出方法如下：

```
for(i=0; i<10; i++)
    printf("%d",a[i]);
```

不能用一个语句输出整个数组，下面的写法是错误的：

```
printf("%d",a);
```

3. 一维数组内存格局

【例 6-1】 以 int a[3]和 char aa[5]为例，说明数组内存的格局。

表 6-1 列出例 6-1 中数组变量内存格局图，其中“==”表示左右两边变量相等，后文同。int 说明数组 a 中每个元素类型为 int，每个元素占用 4 个字节，下标 3 表示元素个数为 3，此数组元素为 a[0]~a[2]。a 同时又是该数组的首地址，是一地址常量。

char 说明数组 aa 中每个元素类型为 char，每个元素占用 1 个字节，下标 5 表示元素个数为 5，此数组元素为 aa[0]~a[4]。aa 同时又是该数组的首地址。

由于 a 和 aa 代表的是内存地址，编译后逻辑地址固定，所以是地址常量，且 a 和 aa 本身代表地址时本身不占内存空间，不像指针变量有专门的 4 个字节来存放变量的内存地址，数组名为地址常量，而指针变量是地址变量。

表 6-1 一维数组内存布局表

一维数组的内存格局，假设变量的堆栈起始地址为 3000			
内存地址	内存	假设的内存值	说 明
3000	a[2]	9	编译完成后，机器代码只认识地址，不认识变量，如 a[2]=9 时，编译后的机器代码为向内存地址 3000~2997 内存里写入数据 9
2999			
2998			
2997			
2996	a[1]	6	a[1]代表内存地址 2996~2993 空间的抽象，&a1 为内存地址 2996。由于此数组类型为 int，每个元素占用 4 个字节，如 a[1]=6，编译完成后，机器代码转换为对相应内存地址的操作，就是向地址为&a[1]（即 2996）相邻 4 个字节的内存里写入 6
2995			
2994			
2993			
2992	a[0]	3	a[0]代表内存地址 2992~2989 内存空间的抽象，其存放的值为 3，此时 a[0]可以与 3 画上等号。&a[0]等于内存地址 2992，同时数组名 a 也代表内存地址 2992，所以 a==&a[0]
2991			
2990			
2989			
2988	aa[4]		编译器通过元素类型确定每个元素的大小，通过元素类型和元素下标确定每个数组元素的内存地址，公式为“数组首地址+下标*单个元素长度”
2987	aa[3]		
2986	aa[2]		
2985	aa[1]		
2984	aa[0]		

4. 一维数组的初始化

给数组赋值的方法除了用赋值语句对数组元素逐个赋值外，还可采用初始化赋值和动态赋值的方法。

数组初始化赋值是指在数组定义时给数组元素赋予初值，数组初始化是在编译阶段进行的，这样将减少运行时间、提高效率。

初始化赋值的一般形式为：

类型说明符 数组名[常量表达式]={值,值.....值};

其中在{ }中的各数据值即为各元素的初值，各值之间用逗号间隔。

例如，inta[10]={ 0,1,2,3,4,5,6,7,8,9 };相当于 a[0]=0;a[1]=1...a[9]=9;。

c 语言对数组的初始化赋值还有以下几点规定：

(1) 可以只给部分元素赋初值。

当{ }中值的个数少于元素个数时，只给前面部分元素赋值。

例如：

```
int a[10]={0,1,2,3,4};
```

表示只给 a[0]~a[4] 五个元素赋值，而后五个元素自动赋值为 0。

(2) 只能给元素逐个赋值，不能给数组整体赋值。

例如，给 10 个元素全部赋 1 值，只能写为：

```
int a[10]={1,1,1,1,1,1,1,1,1,1};
```

而不能写为：

```
int a[10]=1;
```

(3) 如果给全部元素赋值，则在数组说明中，可以不给出数组元素的个数。

例如：

```
int a[5]={1,2,3,4,5};
```

可写为：

```
int a[]={1,2,3,4,5};
```

5. 一维数组应用举例

【例 6-2】 输入 10 个数，并输出最大的数。

arrmax.c 源代码如下：

```
#include <stdio.h>
int main()
{
    int i,max,a[10];
    printf("input 10 numbers:\n");
    for(i=0;i<10;i++)
        scanf("%d",&a[i]);
    max=a[0];
    for(i=1;i<10;i++)
        if(a[i]>max) max=a[i];
    printf("maxmum=%d\n",max);
    return 0 ;
}
```

编译 gcc arrmax.c -o arrmax。

执行 ./arrmax，执行结果如下：

```
input 10 numbers:
3 9 1 900 100 800 700 600 90 0
```

maxmum=900

6.1.3 二维数组

1. 二维数组的定义

二维数组定义的一般形式如下：

类型说明符 数组名[常量表达式 1][常量表达式 2] ；

其中，常量表达式 1 表示第一维下标的长度，常量表达式 2 表示第二维下标的长度。

例如：

```
char a[3][4];
```

说明了一个三行四列的数组，数组名为 a，其下标变量的类型为整型。该数组的下标变量共有 3×4 个，即：

```
a[0][0],a[0][1],a[0][2],a[0][3],
a[1][0],a[1][1],a[1][2],a[1][3],
a[2][0],a[2][1],a[2][2],a[2][3]
```

二维数组在概念上是二维的，即是说其下标在两个方向上变化，下标变量在数组中的位置也处于一个平面之中，而不是像一维数组，只是一个向量。但是，实际的硬件存储器却是连续编址的，也就是说存储器单元是按一维线性排列的，二维数组也是线性存储在内存中的。

二维数组长度计算公式如下：

二维数组长度=存储类型长度*第一维下标*第二维下标

表 6-2 列出了 char a[3][4]的内存布局表，从表中可以看出，二维数组内存空间是连续分配的，数组名和数组的一维下标表示的都是内存地址。

表 6-2 char a[3][4]的内存布局表

一维数组的内存格局，假设变量的堆栈起始地址为-2889		
内存地址	内 存	补充说明
-2889	a[2][3]	
-2990	a[2][2]	
-2991	a[2][1]	
-2992	a[2][0]	此时 a[2]==&a[2][0]==-2992
-2993	a[1][3]	
-2994	a[1][2]	
-2995	a[1][1]	
-2996	a[1][0]	此时 a[1]==&a[1][0]==-2996
-2997	a[0][3]	
-2998	a[0][2]	
-2999	a[0][1]	
-3000	a[0][0]	此时 a==a[0]==&a[0][0]==-3000

2. 二维数组的初始化

二维数组初始化也是在类型说明时，给各下标变量赋以初值。二维数组可按行分段赋值，也可按行连续赋值。

例如，对数组 `a[5][3]`，可用如下两种方法初始化。

① 按行分段赋值可写为：

```
int a[5][3]={ {80,75,92},{61,65,71},{59,63,70},{85,87,90},{76,77,85} };
```

② 按行连续赋值可写为：

```
int a[5][3]={ 80,75,92,61,65,71,59,63,70,85,87,90,76,77,85};
```

这两种赋初值的结果是完全相同的。

对于二维数组初始化赋值还有以下说明：

(1) 可以只对部分元素赋初值，未赋初值的元素自动取 0 值。

```
例如：int a[3][3]={ {1},{2},{3}};
```

是对每一行的第一列元素赋值，未赋值的元素取 0 值。赋值后各元素的值为：

```
1 0 0
2 0 0
3 0 0
```

```
int a [3][3]={ {0,1},{0,0,2},{3}};
```

赋值后的元素值为：

```
0 1 0
0 0 2
3 0 0
```

(2) 如果对全部元素赋初值，则第一维的长度可以不给出。

```
例如：int a[3][3]={1,2,3,4,5,6,7,8,9};
```

```
可以写为：int a[][3]={1,2,3,4,5,6,7,8,9};
```

(3) 数组是一种构造类型的数据，二维数组可以看做是由一维数组的嵌套而构成的，假设一维数组的每个元素都又是一个数组，就组成了二维数组。当然，前提是各元素类型必须相同。根据这样的分析，一个二维数组也可以分解为多个一维数组，C 语言允许这种分解。

如二维数组 `a[3][4]` 可分解为三个一维数组，其数组名分别为 `a[0]`、`a[1]`、`a[2]`，这三个一维数组都有 4 个元素，如一维数组 `a[0]` 的元素为 `a[0][0]`、`a[0][1]`、`a[0][2]`、`a[0][3]`。

必须强调的是，`a[0]`、`a[1]`、`a[2]` 不能当做下标变量使用，它们是数组名，不是一个单纯的下标变量，但 `a[0]`、`a[1]`、`a[2]` 代表数组第二维的首地址。

3. 多维数组定义

C 语言允许构造多维数组，多维数组元素有多个下标，以标识它在数组中的位置，所以也称为多下标变量。多维数组可由二维数组类推而得。

多维数组定义语法形式如下：

类型 数组名[长度 1][长度 2]……[长度 N]；

例如，int m[9][9][8][9]占用空间为 4*9*9*8*9。

4. 二维数组程序举例

【例 6-3】 一个学习小组有 5 个人，每个人有三门课的考试成绩，求全组分科的平均成绩和各科总平均成绩。表 6-3 列出了此 5 人的三门课的考试成绩。

表 6-3 二维数组应用案例表

	张	王	李	赵	周
Math	80	61	59	85	76
C	75	65	63	87	77
Java	92	71	70	90	85

此时可设一个二维数组 a[5][3]，用于存放 5 个人三门课的成绩，再设一个一维数组 v[3] 存放所求得的分科平均成绩，设变量 average 为全组各科总平均成绩。

twoarray.c 源代码如下：

```
#include <stdio.h>
int main()
{
    int i,j,s=0,average,v[3],a[5][3];
    printf("input score\n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<5;j++)
        { scanf("%d",&a[j][i]);
          s=s+a[j][i];}
        v[i]=s/5;
        s=0;
    }
    average =(v[0]+v[1]+v[2])/3;
    printf("math:%d\nc languag:%d\nJava:%d\n",v[0],v[1],v[2]);
    printf("total:%d\n", average );
    return 0 ;
}
```

编译 gcc twoarray.c -o twoarray。

执行 ./twoarray，执行结果如下：

```
input score
80    61    59    85    76
75    65    63    87    77
```

```
92      71      70      90      85
math:72
c languag:73
Java:81
total:75
```

6.1.4 字符数组

1. 字符数组说明

在 C 语言中没有专门的字符串变量，通常用一个字符数组来存放一个字符串。前面介绍字符串常量时，已说明字符串总是以 '\0' 作为串的结束符，因此当把一个字符串存入一个数组时，也把结束符 '\0' 存入了数组，并以此作为该字符串是否结束的标志。有了 '\0' 标志后，就不必再用字符数组的长度来判断字符串的长度了。

C 语言允许用字符串的方式对数组做初始化赋值。

例如：

```
char c[]={ 'c', ' ', 'p', 'r', 'o', 'g', 'r', 'a', 'm' };
```

可写为：

```
char c[]="C program";
```

或去掉 {}, 写为：

```
char c[]="C program";
```

用字符串方式赋值比用字符逐个赋值要多占一个字节，用于存放字符串结束标志 '\0'。上面的后两种数组 c 在内存中的实际存放情况为：

C		p	r	o	g	r	a	m	\0
---	--	---	---	---	---	---	---	---	----

'\0' 是由 C 编译系统自动加上的。由于采用了 '\0' 标志，所以在用字符串赋初值时，一般无须指定数组的长度，而由系统自行处理。

截断字符数组或者增加结束标志方法为：c[6]= '\0' 或 c[6]=0。

2. 字符串处理说明

如果字符数组存放的是字符串，输入可使用 scanf("%s", 字符数组名)，输出可使用 printf("%s", 字符数组名)。

字符串处理需要包含 <string.h> 头文件，需要使用 strcpy、strlen 等字符串函数，具体使用见第 9 章。

3. 字符数组举例

strarray.c 源代码如下：

```
#include <stdio.h>
#include <string.h>
```

```
int    main()
{
    char st[15];
    char st1[15] ;
    printf("input string:\n");
    scanf("%s",st);
    printf("%s\n",st);
    strcpy(st1, st) ;
    printf("cmp=%d\n", strcmp(st, st1)) ;
    return 0 ;
}
```

编译 gcc strarray.c -o strarray。

执行 ./strarray, 执行结果如下:

```
input string:
hello
hello
cmp=0
```

6.1.5 冒泡法排序

冒泡法排序实例是 c 语言中的一个经典算法，实现多个数值的排序。方法是多次循环进行比较，每次比较时将最大数移动到最上面。每次循环时，找出剩余变量里的最大值，然后减小查询范围。这样经过多次循环以后，就完成了对这个数组的排序。冒泡法排序使用了反复循环和比较的算法，执行了下面这些步骤。

- ① 在第一次循环时，拿第一个数与第二个数进行比较，如果第一个数小于第二个数，就用一个中间变量使这两个数交换。这样就使第一个和第二个数从大到小排列。
- ② 再用第一个数与第三个数比较，使这两个数从大到小排列。用同样的方法，用第一个数与后面所有的数进行比较。
- ③ 经过了第一轮循环比较以后，第一个数一定是所有数里面最大的数。
- ④ 进行第二轮比较，这时让第二个数与后面所有的数比较，使这个数是这些数里面的最大数。
- ⑤ 用循环的方法，依次用后面的一个数与这个数后面的所有数进行比较，这样就完成了这些数的从大到小的排序。

图 6-1 是 5 个变量的冒泡法排序原理图，其排序流程如上所述。

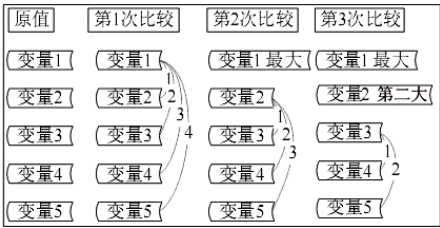


图 6-1 冒泡法排序图

下面的程序是冒泡法排序的例子，冒泡法是通过两次循环比较实现的。

airbubb.c 源代码如下：

```
#include <stdio.h>
int main()
{
    int a[10];
    int i,j,temp;    /*定义循环变量和中间变量*/
    printf("please enter a number:\n"); /*输出提示*/
    for(i=0;i<10;i++)
    {
        scanf("%d",&a[i]); /*进行循环输入变量*/
    }
    for(i=0;i<10;i++)
    {
        for(j=i+1;j<10;j++)
        {
            if(a[i]<a[j])
            {
                temp=a[i];
                a[i]=a[j];
                a[j]=temp;
            }
        }
    }
    for(i=0;i<10;i++)
    {
        printf("%d ",a[i]);
    }
    return 0 ;
}
```

编译 `gcc airbubb.c -o airbubb`。

执行 `./airbubb`，执行结果如下：

```
please enter a number:
3 9 1 900 100 800 700 600 90 0
900 800 700 600 100 90 9 3 1 0
```

6.2 C 语言结构

6.2.1 结构概念

1. 结构存在的意义

存在是合理的，许多事物的存在是在不断解决问题中引入的，当然有更好的方法出现时改变也是合理的。在实际问题中，一组数据往往具有不同的数据类型，例如，在学生登记表中，姓名应为字符型，学号可为整型或字符型，年龄应为整型，性别应为字符型，成绩可为整型或实型。显然不能用一个数组来存放这一组数据，因为数组中各元素的类型和长度都必须一致，以便于编

译系统处理。为了解决这个问题，C 语言中给出了另一种构造数据类型——“结构(structure)”或叫“结构体”，它相当于其他高级语言中的记录。“结构”是一种构造类型，它是由若干“成员”组成的，每一个成员可以是一个基本数据类型或者又是一个构造类型。结构既是一种“构造”而成的数据类型，那么在说明和使用之前必须先定义它，也就是先构造它，如同在声明和调用函数之前要先定义函数一样。

定义一个结构的一般语法形式如下：

```
struct 结构名
{
    成员表列
};
```

成员表列由若干个成员组成，每个成员都是该结构的一个组成部分。对每个成员也必须做类型说明，其形式如下：

类型说明符 成员名；

成员名的命名应符合标识符的书写规定。结构定义举例说明如下：

```
struct stu
{
    int num;
    char name[20];
    char sex;
    float score;
};
```

在这个结构定义中，结构名为 `stu`，该结构由 4 个成员组成。第一个成员是 `num`，为整型变量；第二个成员是 `name`，为字符数组；第三个成员是 `sex`，为字符变量；第四个成员是 `score`，为实型变量。应注意在括号后的分号是不可缺少的。结构定义之后，即可进行结构变量定义，凡定义为结构 `stu` 的变量都由上述 4 个成员组成。由此可见，结构是一种复杂的数据类型，是数目固定、类型不同的若干有序变量的集合。

2. 结构概念小结

结构是一种构造数据类型。

结构的用途是把不同类型的数据组合成一个整体，是自定义数据类型。

结构类型定义语法形式如下：

```
struct      [结构名]
{
    类型标识符    成员名;
    类型标识符    成员名;
    .....
}[变量名 1, 变量名 2……];
```

定义结构变量方法为：`struct 结构名 变量名。`

结构变量引用规则为：结构变量不能整体引用，只能引用变量成员，引用方式为“结构变量名.成员名”，可以将一个结构变量赋值给另一个结构变量，结构嵌套时需要逐级引用。

6.2.2 结构变量

1. 结构变量有以下三种定义方法

(1) 先定义结构，再定义结构变量：

```
struct stu
{
    int num;
    char name[8];
    char sex;
    float score;
};
struct stu boy1,boy2;
```

(2) 在定义结构类型的同时定义结构变量：

```
struct stu
{
    int num;
    char name[8];
    char sex;
    float score;
}boy1,boy2;
```

(3) 直接定义结构变量：

```
struct
{
    int num;
    char name[8];
    char sex;
    float score;
}boy1,boy2;
```

第三种方法与第二种方法的区别在于第三种方法中省去了结构名，而直接给出结构变量。三种方法中说明的 boy1、boy2 变量都具有如图 6-2 所示的结构。

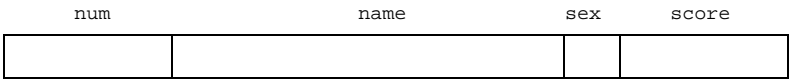


图 6-2 结构变量组成图

2. 结构变量内存布局

所有 C 语言中的变量经过编译后，都会转换为对内存线性地址的操作。各种变量的引入是面向人类思维进行的，方便人们高效的编程，而 C 语言源代码经过 gcc 编译后，形成执行码时，所有变量与符号失去意义，变量与符号转变为对相应内存地址的操作。int 类型代表着 4 个字节相应地址的内存空间的抽象，而结构变量也是代表着一片内存空间的抽象（结构名不代表内存地址），

所以相同类型结构变量与 `int` 类型变量一样可以用“=”号直接赋值（如 `boy2=boy1`），两个结构变量用“=”号赋值代表两片相同大小空间的复制。

内存的物理地址是线性的、一维的，而结构变量往往代表二维或多维。这种二维或多维结构方便人们编程时对变量进行管理，是一种面向人们思维的逻辑结构，但结构变量最终物理表现在内存中还是一维线性结构。

下面以上面定义的 `boy1` 来说明结构变量在内存中的存储方式。表 6-4 列出了结构变量 `boy1` 的内存布局，此时 `boy1` 代表着-19~0 这片内存空间的抽象。`&boy1` 代表结构变量的地址，即为 0。结构成员（如 `boy1.num`）其实也代表着对应地址内存空间的抽象，对结构成员的操作，编译器最后会转化为对相应内存地址的操作。

表 6-4 boy1 结构变量内存布局表

假定栈空间起始地址为 0		
内存地址	内 存	说 明
0	boy1.num	
-1		
-2		
-3		
-4	boy1.name[7]	
-5	boy1.name[6]	
-7	boy1.name[5]	
-8	boy1.name[4]	
-9	boy1.name[3]	
-10	boy1.name[2]	
-11	boy1.name[1]	
-12	boy1.name[0]	body.name 表示内存地址-12
-13	boy1.sex	
-14		float 为 4 个字节，需要到-16 处才能对齐，所以空两个字节
-15		
-16	boy1.score	
-17		
-18		
-19		

3. 结构变量成员的引用

表示结构变量成员引用的一般形式如下：

结构变量名.成员名

例如：

boy1.num

boy2.sex

即第一个人的学号

即第二个人的性别

如果成员本身又是一个结构，则必须逐级找到最低级的成员才能引用。

4. 结构变量的赋值

结构变量的赋值就是给各成员赋值，可用输入语句或赋值语句来完成。

【例 6-4】 给结构变量赋值并输出其值。

stru1.c 源代码如下：

```
#include <stdio.h>
int main()
{
    struct stu
    {
        int num;
        char name[20];
        char sex;
        float score;
    } boy1,boy2;
    boy1.num=102;
    boy1.name="Zhang ping";
    printf("input sex and score\n");
    scanf("%c %f",&boy1.sex,&boy1.score);
    boy2=boy1;
    printf("Number=%d\nName=%s\n",boy2.num,boy2.name);
    printf("Sex=%c\nScore=%f\n",boy2.sex,boy2.score);
    return 0 ;
}
```

编译 gcc stru1.c -o stru1。

执行 ./stru1，执行结果如下：

```
input sex and score
M 90
Number=102
Name=Zhang ping
Sex=M
Score=90.000000
```

5. 结构变量的初始化

【例 6-5】 对结构变量初始化。

stru2.c 源代码如下：

```
#include <stdio.h>
int main()
{
    struct stu
    {
        int num;
        char *name;
        char sex;
```

```
float score;
}boy2,boy1={102,"Zhang ping",'M',78.5};
boy2=boy1;
printf("Number=%d\nName=%s\n",boy2.num,boy2.name);
printf("Sex=%c\nScore=%f\n",boy2.sex,boy2.score);
return 0 ;
}
```

编译 gcc stru2.c -o stru2。

执行 ./stru2，执行结果如下：

```
Number=102
Name=Zhang ping
Sex=M
Score=78.500000
```

6. 结构数组的定义

结构数组的定义与初始化方法如下：

```
struct stu
{
    int num;
    char name[20];
    char sex;
    float score;
}boy[5]={
    {101,"Li ping","M",45},
    {102,"Zhang ping","M",62.5},
    {103,"He fang","F",92.5},
    {104,"Cheng ling","F",87},
    {105,"Wang ming","M",58},
};
```

当对全部元素做初始化赋值时，也可不给出数组长度。

引用结构数组成员的方法：结构[下标].成员变量，如 boy[0].num、boy[1].name。

6.3 指针

指针——C 语言的精华，它在 C 语言中表现得最优秀也最危险。

6.3.1 指针概念

1. 指针概述

内存中每个字节有一个编号，即地址。

变量是对数据存储内存空间的抽象，一般变量（如 int 等）是对变量的直接访问，而指针变量是对变量的间接访问。

指针变量说明此量为一变量，变量需要占用空间，同时指针即内存地址，结合起来就是内存地址变量，即专门用来存放内存地址的变量，该变量存放另一变量的内存地址。指针变量类型指的是指针变量指向目标变量的类型。

指针变量本身没有类型，所有指针变量都占用四个字节的空間，这个四个字節空間里存放的是内存地址，如 `int *p1`、`char *p2`、`double *p3` 中的指针变量 `p1`、`p2`、`p3` 都占用四个字节的内存空间。

理解指针变量指向的变量，`int *p1` 中 `p1` 为指针变量，`*p1` 为指针变量指向的变量。指针变量只能指向定义时所规定类型的变量，此时 `p1` 只能指向 `int` 型变量，即给 `p1` 赋值时只能赋 `int` 型变量的地址。指针变量定义后，变量值不确定，应用前必须先赋值。

`&`与`*`运算符互为逆运算，`*`表示取指针变量所指向变量的内容，`&`表示取变量的地址。假设 `i_pointer` 为指针变量，它的内容是地址量，`*i_pointer` 为指针变量指向的变量，它的内容是数据，`&i_pointer` 为指针变量本身内存的地址。

存放变量地址的变量是指针变量。`int *p1` 说明 `p1` 是指针变量，该指针变量名为 `p1`，`p1` 的值为内存地址。`*p1` 是 `p1` 所指向的变量，`int` 说明指针变量指向变量的类型，即`*p1` 的类型，而不是 `p1` 的类型。

数组名表示数组首地址，是地址常量。

数组名作为函数参数，是地址传递，数组名和指针变量作为函数传递参数时可以通用，数组名作为形参时，被调用函数将数组名当做指针变量处理。

2. 指针变量与其所指向的变量之间的关系

图 6-3 给出了指针变量与其所指向变量之间的关系图， \Leftrightarrow 符号表示等价，其中变量 `i` 的地址为 2000，`i_pointer` 为指针变量，`*i_pointer` 为指针变量所指向的变量。

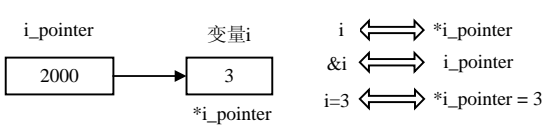


图 6-3 指针变量与其所指向变量之间的关系图

3. 指针变量定义

图 6-4 给出了指针变量定义图，说明了定义时各字段的含义。

指针变量定义的一般形式如下：

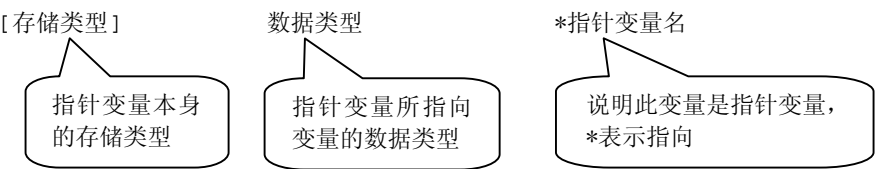


图 6-4 指针变量定义图

指针变量定义说明：

```
int *p1;
float *p2;
```

对上述指针变量，具体解释如下：

- ① 指针变量名是 p1、p2，不是 *p1、*p2。
- ② 指针变量只能指向定义时所规定类型的变量，即 p1 只能指向 int 变量的地址，p2 只能指向 float 变量的地址。
- ③ 指针变量定义后，变量值不确定，使用前必须先赋值。

4. 指针的引用

指针变量的赋值只能赋予地址，绝不能赋予任何其他数据，否则将引起错误。在 c 语言中，变量的地址是由编译系统分配的，对用户完全透明，用户不知道变量的具体地址。有如下两个与指针有关的运算符：

- ① &：取地址运算符。
- ② *：指针运算符（或称“间接访问”运算符）。

c 语言中提供了地址运算符&来表示变量的地址，其一般形式为“&变量名”，如&a 表示变量 a 的地址，&b 表示变量 b 的地址，变量本身必须预先进行定义。

设有指向整型变量的指针变量 p，如要把整型变量 a 的地址赋予 p，可以有以下两种方式：

- ① 指针变量初始化的方法如下：

```
int a;
int *p=&a;
```

- ② 对指针变量赋值的方法如下：

```
int a;
int *p;
p=&a;
```

5. 指针引用图解

指针变量和一般变量一样，存放在它们之中的值是可以改变的，也就是说可以改变它们的指向，假设：

```
char i,j,*p1,*p2;
i='a';
j='b';
p1=&i;
p2=&j;
```

上述语句建立了如图 6-5 所示的联系：

在这里，-100 为 i 的内存地址（即&i），-101 为 j 的内存地址（即&j）。

赋值 p2=p1，就使 p2 与 p1 指向同一对象 i，此时*p2 就等于 i，而不是 j，如图 6-6 所示。

如果执行如下表达式 *p2=*p1，则表示把 p1 指向的内容赋给 p2 所指的内存区域，此时就如图 6-7 所示。

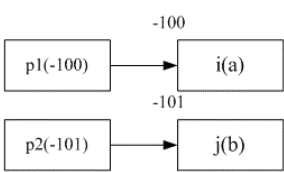


图 6-5 指针引用图 1

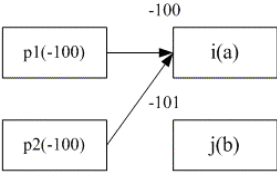


图 6-6 指针引用图 2

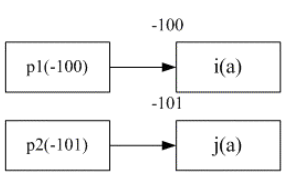


图 6-7 指针引用图 3

表 6-5 列出了上述变量在栈空间上的内存布局，当 p2=p1，就是把 p2 的值改为-100，而*(-100)代表变量 i。当*p2=*p1，即*(-101)=*(-100)，是对两个内存地址指向的空间进行赋值，相当于 j=i，而 p2 本身的值并没有改变。*p1 是地址-100 内存空间的抽象，*p2 是地址-101 内存空间的抽象，而 p1 是地址-104~-107 内存空间的抽象，p2 是地址-108~-111 内存空间的抽象。

表 6-5 上述变量在栈空间上的内存布局表

假设变量的堆栈起始地址为-100,()中代表变量的值		
内存地址	内 存	说 明
-100	i (a)	
-101	j (b)	
-102		由于整型变量需要对齐，地址必须被 4 整除，对于指针本身，编译器把它当做整型数据处理，所以空两个字节
-103		
-104	p1 (-100)	此时 p1 等于-100，而-100 是变量 i 的内存地址，*(-100)代表指向内存地址-100 的变量，即变量 i
-105		
-106		
-107		
-108	p2 (-101)	
-109		
-110		
-111		

6. 理解指针变量

假设：

```
int a=3, *p1;
char *p2, aa='X';
p1=&a ;
p2=&aa ;
```

表 6-6 列出了上述变量在内存中的布局并给出其详细说明。

表 6-6 指针变量内存布局表

假设变量的堆栈起始地址为 3000, ()中为变量的值		
内存地址	内 存	说 明
3000	a (3)	变量是相应地址内存空间的抽象，变量 a 是内存地址 3000~2997 这片内存空间的抽象，其值初始化等于 3，int 型说明此变量占用空间大小为 4 个字节。&a 等于变量 a 的内存地址，即 3000
2999		
2998		
2997		
2996	p1 (3000)	p1 是一个指针变量（即地址变量），保存的值为 3000。*p1 是指针变量 p1 所指向的变量，此时*p1 等于*3000，即地址 3000 所指向的变量（即内存地址 3000 所存的变量值，也就是变量 a），*p1 此时可以与变量 a 画等号，值为 3
2995		
2994		
2993		
2992	p2 (2888)	char *p2 说明指针变量指向的变量类型为 char，p2 等于 2888，是变量 aa 的内存地址，*p2 等于*2888，即变量 aa，其值为 x
2991		
2990		
2889		
2888	aa(X)	aa 是一个 char 类型的变量，占用一个字节的内存空间

对上述指针变量的解释如下：

① 指针变量是一个特殊的变量，它里面存储的数值是内存里的一个地址。不管指针变量前面的类型如何，32 位机器中指针变量都一律占用内存空间 4 个字节，这 4 个字节存放的就是其他变量的内存地址。所以指针变量可以理解成内存地址变量，*可理解为指向。

② 理解指针变量和指针变量指向的变量，上述 p1、p2 是指针变量，*p1、*p2 是指针变量指向的变量。

③ 理解指针变量的值和指针变量指向变量的值，指针变量的值为 p1 等于 3000、p2 等于 2888。指针变量指向变量的值即内存地址指向变量的值，*p1 即*3000，*3000 等于变量 a，即 3；*p2 即*2888，*2888 等于变量 aa，即字符 x。

理解指针变量类型，通常所说指针变量类型是指针变量所指向变量的类型。*p1 代表 p1 指向的类型为 int 型，占用 4 个字节，*p2 代表 p2 指向的类型为 char 型，占用 1 个字节的内存空间。

理解指针变量本身类型，所有指针变量本身占用 4 个字节，存放内存地址。本身类型在编程中毫无意义，可以理解成无类型，也可以理解成是 4 个字节的整型数据。

int a 说明 a 是整型变量，其 a 有两重限定含义，其一为变量，其二类型为整型，两者合在一起即为整型变量；char aa 说明 aa 是字符变量。int *p 中的*说明 p 是指针变量，指针是变量的限定词，说明此变量的类型是指针（内存地址），int 说明此指针变量所指向变量的类型为 int。

通过指针变量是对目标变量的间接操作。例如，我们找小强，既可以通过直接操作找小强，也可以通过间接操作，先找到小强的爸爸，通过小强爸爸的指引找到小强。上述的 a=3 是直接操

作，而 `p1=&a`、`*p1=3` 则是间接操作，虽然两者结果相同，但过程不同。直接操作是直接在相应内存地址上读写数据，而间接操作则需两次或多次读写内存，间接操作是 CPU 首先从内存中读到变量地址，再根据变量地址从内存中读写数据。

6.3.2 sizeof、void、const 说明

1. sizeof 运算符

`sizeof` 运算符是 C 语言的关键字，用于求一个对象所占用的字节数。使用 `sizeof` 运算符计算对象大小是一个良好的编程习惯。

【例 6-6】 利用 `sizeof` 更深刻地理解指针变量。

`sizeof.c` 源代码如下：

```
#include <stdio.h>
int main()
{
    int *p_int ;
    char *p_char ;
    float *p_float ;
    double *p_double ;
    int var_int ;
    char var_char ;
    float var_float ;
    double var_double ;
    struct stu
    {
        int num;
        char name[8];
        char sex;
        float score;
    } ;
    struct stu *p_str, var_str ;
    p_int=&var_int ;
    p_char=&var_char ;
    p_float=&var_float ;
    printf("p_int=%d, *p_int=%d\n", sizeof(p_int), sizeof(*p_int) ) ;
    printf("p_char=%d, *p_char=%d\n", sizeof(p_char), sizeof(*p_char) ) ;
    printf("p_float=%d, *p_float=%d\n", sizeof(p_float), sizeof(*p_float) ) ;
    printf("p_double=%d, *p_double=%d\n", sizeof(p_double), sizeof(*p_double) ) ;
    printf("p_str=%d, *p_str=%d\n", sizeof(p_str), sizeof(*p_str) ) ;
    printf("int length=%d\n", sizeof(int) ) ;
    printf("char length=%d\n", sizeof(char) ) ;
    printf("float length=%d\n", sizeof(float) ) ;
    printf("double length=%d\n", sizeof(double) ) ;
    printf("struct stu length=%d\n", sizeof(struct stu) ) ;
    return 0 ;
}
```

编译 `gcc sizeof.c -o sizeof`。

执行 `./sizeof`，执行结果如下：

```
p_int=4, *p_int=4
p_char=4, *p_char=1
p_float=4, *p_float=4
p_double=4, *p_double=8
p_str=4, *p_str=20
int length=4
char length=1
float length=4
double length=8
struct stu length=20
```

从上面运行结果可以看出，指针变量无论类型是什么，都只占用四个字节的内存空间，而指针变量指向的变量就是变量类型定义的大小。

2. void 类型指针

void 表示无类型，通常用来修饰指针变量，使用时需要强制转换。

```
void *p ;
p=(int *)malloc(sizeof(int)*100) ;
```

3. const 关键字说明

类型声明中 const 用来修饰一个常量。修饰时有以下情况，请读者理解掌握。

```
const int nValue; //nValue 是 const
const char *pContent; // *pContent 是 const, pContent 可变
const (char *) pContent; // pContent 是 const, *pContent 可变
char* const pContent; // pContent 是 const, *pContent 可变
const char* const pContent; // pContent 和 *pContent 都是 const
```

6.3.3 指针变量作为函数参数

【例 6-7】 指针变量作为函数参数的运行机理

swap_p.c 源代码如下：

```
#include <stdio.h>
int swap(int *p1,int *p2)
{
    int temp;
    temp=*p1;
    *p1=*p2;
    *p2=temp;
    return 0 ;
}
int main()
{
    int a,b;
    int *pointer_1,*pointer_2;
    scanf("%d,%d",&a,&b);
    pointer_1=&a;pointer_2=&b;
    if(a<b) swap(pointer_1,pointer_2);
    printf("%d,%d\n",a,b);
}
```

```
    return 0 ;  
}
```

编译 gcc swap_p.c -o swap_p。

执行 ./swap_p，执行结果如下：

```
80,90  
90,80
```

对上述程序的说明：

- ① 变量的赋值只不过是把内存一个地址上的值复制到另一个地址而已。
- ② 实参 pointer_1 传给形参 p1 后，由于 pointer_1 的值为&a，所以在 swap 函数中对 *p1 的操作，其实是对*(&a)的操作，即对 a 的操作。
- ③ 一个程序中指针变量可以指向该执行空间堆栈段任何内存地址，所以即使在 swap 函数中，也能实现对该程序中任何变量进行操作，只要形参传入该变量地址即可完成对该地址上变量的操作。例如，*p1 操作的是 main 函数中的 a 变量，因为 p1 的值等于 a 变量的地址。

指针变量可以指向执行程序（进程）整个堆栈空间，指针变量的值（内存地址）不正确或越界会造成程序执行错误或 coredump，指针变量的值不正确或越界是常见的编程错误。

表 6-7 列出了 swap_p 程序运行内存堆栈空间表，查看此表可以更好地理解指针作为函数形参时的调用原理，假设栈段起始值为 30000，下表圆括号 () 中的值是变量的值。

表 6-7 swap_p 程序运行时内存堆栈表

数据段说明	内存地址	说 明
栈段	30000	main 函数返回值存放空间
	29996	a(80) 此时 a 等于 80，&a 等于 29996，*(&a)等于 a
	29992	b(90)
	29988	pointer1 (29996)
	29984	pointer2(29992)
	29980	swap 函数返回值存放空间 (0)
	29976	p1 (29996)
	29972	p2 (29992)
	29968	temp
堆段
bss 段
静态数据段
代码段		1. swap 函数地址入口 2. 在栈顶分配整型变量 temp 空间 3. 执行 temp=*p1，即 temp=*29996==a==80 4. 执行 *p1=*p2，即*29996=*29992，即*(&a)=*(&b)，即 a=b==90，即 a=90 5. 执行 *p2=temp，即*29992=80，即*(&b)=80，即 b=80 6. 返回 0 给函数返回值空间，即内存地址 29980~29977 处

续表

数据段说明	内存地址	说 明
代码段		7. 函数结束返回。 8. main 函数地址入口 9. 在栈顶分配 main 函数返回值、a、b、pointer1、pointer2 变量空间 10. 从屏幕上读取两个数字分别存放到 a、b 变量的内存空间里，此时&a 等于内存地址 29996，而 a 代表内存地址 29996~29993 内存空间抽象，读入变量 a 的值存放到内存地址 29996~29993 的内存空间里。变量 b 同理 11. 执行 pointer_1=&a，即 pointer_1=29996 12. 执行 pointer_2=&b，即 pointer_2=29992 13. 如果 a<b，调用 swap(pointer_1, pointer_2)函数，在栈顶分配 swap 返回值空间、p1 和 p2 变量空间，并将实参 pointer_1 的值复制给 p1，实参 pointer_2 的值复制给 p2。然后跳转到 1（swap 函数入口）的地方去执行 14. 打印 a、b 的值到屏幕上 15. 返回值 0 写入到 main 返回值存放空间，即内存地址 30000~29997 内存空间处 16. 程序执行完成，释放所有内存空间

6.3.4 指针的运算

1. 加减算术运算

对于指向数组的指针变量，可以加上或减去一个整数 n。假设 pa 是指向数组 a 的指针变量，则 pa+n、pa-n、pa++、++pa、pa--、--pa 运算都是合法的。

指针变量的加减运算只能对数组指针变量进行，对指向其他类型变量的指针变量做加减运算是毫无意义的。

指针变量加减偏移量与其前面定义的类型密切相关，int 类型指针变量加 1 地址偏移 4 位，char 类型指针变量加 1 偏移 1 位。

【例 6-8】 指针变量加减运算地址的偏移量。

p_airth.c 源代码如下：

```
#include <stdio.h>
int main()
{
    int x[10], *px ;
    char y[10], *py ;
    px=x ;
    py=&y[0] ;
    printf("px=%d, py=%d\n", px, py) ;
    px++ ;
    py++ ;
    printf("px1=%d, py1=%d\n", px, py) ;
    return 0;
}
```

编译 `gcc p_airth.c -o p_airth`。

执行 `./p_airth`，执行结果如下：

```
px=-1074644316, py=-1074644266
px1=-1074644312, py1=-1074644265
```

2. 两个指针变量之间的运算

只有指向同一数组的两个指针变量之间才能进行运算，否则运算毫无意义。两个指针变量之间的运算有下面两种情况。

① 两指针变量相减：两指针变量相减所得之差除以指针变量指向变量的类型长度是两个指针所指数组元素之间相差的元素个数。

```
int a[10], *p1, *p2 ;
p1=&a[0];
p2=&a[7];
```

此时 $(p2-p1)/4$ 等于 7，因为 `int` 占用 4 个字节，数组在内存中是顺序存储的。

② 两指针变量进行关系运算：指向同一数组的两指针变量进行关系运算可表示它们所指数组元素之间的关系。

例如：

`pf1==pf2` 表示 `pf1` 和 `pf2` 指向同一数组元素。

`pf1>pf2` 表示 `pf1` 处于高地址位置。

`pf1<pf2` 表示 `pf2` 处于低地址位置。

指针变量还可以与 0 比较，设 `p` 为指针变量，则 `p==0` 表明 `p` 是空指针，它不指向任何变量，`p!=0` 表示 `p` 不是空指针，空指针是对指针变量赋予 0 值而得到的。

例如：

```
#define NULL 0
int *p=NULL;
```

对指针变量赋 0 值和不赋值是不同的。指针变量未赋值时，可以是任意值，是不能当右值使用的，否则将造成意外错误。而指针变量赋 0 值后，则是安全的，它不指向任何具体的变量，指针变量不使用时应赋 0 值。

3. 指针的赋值运算

指针的赋值运算有下面 6 种情形。

① 指针变量初始化赋值，前面已经介绍过。

② 把一个变量的地址赋予指向相同数据类型的指针变量。

```
int a,*pa;
pa=&a;    /*把整型变量 a 的地址赋予整型指针变量 pa*/
```

③ 把一个指针变量的值赋予指向相同类型变量的另一个指针变量。

```
int a,*pa=&a,*pb;
pb=pa;    /*把 a 的地址赋予指针变量 pb*/
```

由于 pa、pb 均为指向整型变量的指针变量，因此可以相互赋值。

④ 把数组的首地址赋予指向数组的指针变量。

```
int a[5],*pa;
pa=a; /*数组名表示数组的首地址，故可赋予指向数组的指针变量 pa*/
```

也可写为：

```
pa=&a[0]; /*数组第一个元素的地址也是整个数组的首地址，也可赋予 pa*/
```

当然也可采取初始化赋值的方法：

```
int a[5],*pa=a;
```

⑤ 把字符串的首地址赋予指向字符类型的指针变量。

```
char *pc;
pc="C Language"; /*编译程序先开辟静态数据区存放字符串，然后指针变量 pc 指向它*/
```

或用初始化赋值的方法写为：

```
char *pc="C Language";
```

此时相当于：

```
static char st[]={"C Language"}, *pc=&st ;
```

⑥ 把函数的入口地址赋予指向函数的指针变量。

```
int (*pf)();
pf=f;    /*f 为函数名*/
```

其实函数也有其入口地址，指针变量指向函数地址入口，就代表对此函数的执行。

6.3.5 指向数组的指针变量

1. 数组与指针的关系

在许多程序员眼里，数组与指针有说不清、道不明的差别，剪不断、理还乱的关系。它们俩什么时候是等价，什么时候又不等价，为什么许多时候可以等价使用，又有一些时候判若两人呢？为了揭开它们俩差别的神秘面纱，还得从 CPU 这个关键“人物”说起，CPU 看待变量，只不过是段内存空间的抽象而已。数组变量也好，指针变量也罢，最终都会转化为对内存地址的操作，对于数组与指针，是编译器让这两位“兄弟”在右值时等价。又因为数组名是地址常量（数组名代表数组首地址，不能改变，编译时确定），而指针变量是地址变量（有 4 个字节的内存空间，是

变量理所当然执行时可以不断改变),所以在左值时只能有指针变量可以使用,而数组名是地址常量,不占内存空间,当然就不能用其他变量或常量对其赋值。

一个变量有一个地址,一个数组包含若干元素,每个数组元素都在内存中占用一定的存储单元,它们都有相应的内存地址,而且数组元素是按顺序占用连续的内存空间。所谓数组的指针是指数组的起始地址,数组元素的指针是指某个数组元素的地址。

2. 一级指针变量与一维数组的关系

假设定义: `int *p` 与 `int q[10]`,下面详细说明两者的联系与区别,“ \Leftrightarrow ”表示两者等价。

数组名是指针(地址)常量。

当 $p=q$ 时, $p+i$ 是 $q[i]$ 的地址。

数组元素的表示方法有下标法和指针法两种,若 $p=q$,则 $p[i] \Leftrightarrow q[i] \Leftrightarrow *(p+i) \Leftrightarrow *(q+i)$ 。

形参是数组时实质上是形参变量是指针变量,即 `int q[]` \Leftrightarrow `int *q`,调用时该形参变量占用 4 个字节,存放数组的首地址。

在定义指针变量(不是形参)时,不能把 `int *p` 写成 `int p[]`。

系统只给 p 分配 4 字节的内存区(32 位机器),而给 q 分配 $4*10$ 字节的内存区。

3. 指针变量与数组名的等价使用

如果指针变量 p 已指向数组中的一个元素,则 $p+1$ 指向同一数组中的下一个元素。引入指针变量后,就可以用下标法和指针法两种方法来访问数组元素了。

```
int a[10], p=&a[0];
```

此时 p 的初值为 $\&a[0]$,则:

1) $p+i$ 和 $a+i$ 就是 $a[i]$ 的地址,或者说它们指向 a 数组的第 i 个元素。

图 6-8 给出了指针变量与数组名等价使用图。

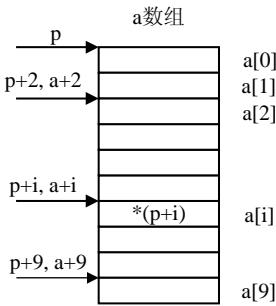


图 6-8 指针变量与数组名等价使用图

2) $*(p+i)$ 或 $*(a+i)$ 就是 $p+i$ 或 $a+i$ 所指向的数组元素,即 $a[i]$ 。

例如，*(p+5)或*(a+5)就是 a[5]。

3) 指向数组的指针变量也可以带下标，如 p[i]与*(p+i)等价。

根据以上叙述，引用一个数组元素可以有如下两种方法：

① 下标法：用 a[i]形式访问数组元素。

② 指针法：若 p=a，可采用*(a+i)或*(p+i)形式访问数组元素，其中 a 是数组名，p 是指向数组的指针变量。

注意指针变量可以实现本身值的改变，如 p++是合法的，而 a++是错误的。因为 a 是数组名，是地址常量，不能进行自加减运算。

【例 6-9】 指针变量与数组名的等价使用。

p_array.c 源代码如下：

```
#include <stdio.h>
int main()
{
    int a[10]={0,1,2,3,4,5,6,7,8,9} ;
    int *pa ;
    pa=a ;
    printf("a[9]=%d\n", a[9]) ;
    printf("pa[9]=%d\n", pa[9]) ;
    printf("(*(a+9))=%d\n", *(a+9)) ;
    printf("(*(pa+9))=%d\n", *(pa+9)) ;
    return 0 ;
}
```

编译 gcc p_array.c -o p_array。

执行 ./p_array，执行结果如下：

```
a[9]=9
pa[9]=9
*(a+9)=9
*(pa+9)=9
```

从上面的执行结果可以看出，这四种表示法在右值时是等价的。假设 a 代表地址 3000，上面四种表示法经编译过后都是*(3000+9*4)，都代表 a[9]。

6.3.6 数组名作为函数参数

1. 形参是数组的本质特征

形参是数组时实质上形参变量是指针变量，即 int q[]⇔int *q，调用时该形参变量 q 占用 4 个字节，存放数组的首地址。

```
f(int x[],int n)等同于 f(int *x,int n)
```

2. 形参是数组名时，与指针可等价使用

fun_array.c 源代码如下：

```
#include <stdio.h>
// void inv(int *x, int n)
void inv(int x[],int n) /*形参 x 是数组名*/
{
    int temp,i,j,m=(n-1)/2;
    for(i=0;i<=m;i++)
    {j=n-1-i;
        temp=x[i];x[i]=x[j];x[j]=temp;}
    // temp=*(x+i); *(x+i)=*(x+j); *(x+j)=temp; }
    return;
}
int main()
{int i,a[10]={3,7,9,11,0,6,7,5,4,2};
    printf("The original array:\n");
    for(i=0;i<10;i++)
        printf("%d,",a[i]);
    printf("\n");
    inv(a,10);
    printf("The array has benn inverted:\n");
    for(i=0;i<10;i++)
        printf("%d,",a[i]);
    printf("\n");
    return 0 ;
}
```

编译 gcc fun_array.c -o fun_array。

执行 ./fun_array，执行结果如下：

```
The original array:
3,7,9,11,0,6,7,5,4,2,
The array has benn inverted:
2,4,5,7,6,0,11,9,7,3,
```

可以看出，数组名作为形参时可以与指针变量互换。

此时 void inv(int *x, int n)与 void inv(int x[],int n)等价；temp=x[i];x[i]=x[j];x[j]=temp 与 temp=*(x+i); *(x+i)=*(x+j); *(x+j)=temp 等价

上面两两组合可构成四种情形。

3. 多维数组数组名含义

若有 int a[3][4]，图 6-9 给出了二维数组 a 的地址，每个方格中是多维数组元素的地址，此时 a、a[0]、a[1]、a[2]都是地址常量，编译时确定，其值说明如下。

```
a==a[0]==&a[0][0]==1000
a+1==a[1]==&a[1][0]==1008
a+2==a[2]==&a[2][0]==1016
```

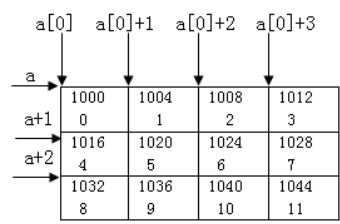


图 6-9 多维数组地址图

6.3.7 函数指针变量

在 C 语言中，一个函数总是占用一段连续的内存区，而函数名就是该函数所占内存区的首地址。可以把函数的这个首地址（或称入口地址）赋予一个指针变量，使该指针变量指向该函数，然后通过指针变量就可以找到并调用这个函数。这种指向函数的指针变量称为“函数指针变量”。

函数指针变量定义的一般形式如下：

类型说明符 (*指针变量名)();

其中，“类型说明符”表示被指函数的返回值类型。“(* 指针变量名)”表示定义的是指针变量，最后的空圆括号表示指针变量所指的是一个函数。

例如：

```
int (*pf)();
```

pf 是一个指向函数入口的指针变量，该函数的返回值（函数值）是整型。

【例 6-10】 函数指针变量举例。

p_fun.c 源代码如下：

```
#include <stdio.h>
int max(int a,int b)
{
    if(a>b)return a;
    else return b;
}
int main()
{
    int max(int a,int b);
    int (*pmax)();
    int x,y,z;
    pmax=max;
    printf("input two numbers:\n");
    scanf("%d%d",&x,&y);
    z=(*pmax)(x,y);
    printf("maxmum=%d\n",z);
    return 0 ;
}
```

编译 gcc p_fun.c -o p_fun。

执行 `./p_fun`, 执行结果如下:

```
input two numbers:
2000 3000
maxmum=3000
```

6.3.8 返回指针类型函数

所谓函数类型是指函数返回值的类型。在 C 语言中允许一个函数的返回值是一个指针 (即地址), 这种返回指针值的函数称为指针型函数。

定义指针型函数的一般形式如下:

```
类型说明符 *函数名(形参表)
{
    ..... /*函数体*/
}
```

其中, 函数名之前加了 “*” 号表明这是一个指针型函数, 即返回值是一个指针, 类型说明符表示返回的指针变量所指向的数据类型。

【例 6-11】 本程序是通过指针函数, 输入一个 1~7 之间的整数, 输出对应的星期名。

`p_week.c` 源代码如下:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int i;
    char *day_name(int n);
    printf("input Day No:\n");
    scanf("%d",&i);
    if(i<0) exit(1);
    printf("Day No:%2d-->%s\n",i,day_name(i));
    return 0 ;
}
char *day_name(int n){
    static char *name[]={ "Illegal day",
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday",
        "Sunday"};
    return((n<1||n>7) ? name[0] : name[n]);
}
```

编译 `gcc p_week.c -o p_week`。

执行 `./p_week`, 执行结果如下:

```
input Day No:
```

3
Day No: 3-->Wednesday

6.3.9 指向指针的指针

如果一个指针变量存放的又是另一个指针变量的地址，则称这个指针变量为指向指针的指针变量。

通过指针访问变量称为间接访问。由于指针变量直接指向变量，所以称为“单级间址”，而如果通过指向指针的指针变量来访问变量则构成“二级间址”，图 6-10 给出二维指针变量实现图。

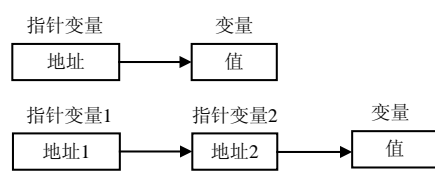


图 6-10 二维指针变量实现图

【例 6-12】 一个指针数组元素指向数据的简单例子。

p2_array.c 源代码如下：

```
#include <stdio.h>
int main()
{
static int a[5]={1,3,5,7,9};
    int *num[5]={&a[0],&a[1],&a[2],&a[3],&a[4]};
    int **p,i;
    p=num;
    for(i=0;i<5;i++)
    {
printf("%d\t",**p);
p++;
}
    printf("\n") ;
    return 0 ;
}
```

编译 gcc p2_array.c -o p2_array。

执行 ./p2_array, 执行结果如下：

1 3 5 7 9

6.3.10 结构指针

1. 结构指针变量定义

一个指针变量用来指向一个结构变量时，称之为结构指针变量。结构指针变量中的值是所指向的结构变量首地址，通过结构指针即可访问该结构变量，这与数组指针和函数指针的情况是相同的。

结构指针变量定义语法形式如下：

```
struct 结构名 *结构指针变量名
```

2. 结构作为函数形参特点

用结构变量的成员作为形参或结构变量作为形参都为值传递，效率低，用结构指针变量作为参数为地址传递。

结构指针变量也是地址变量，占用 4 个字节的内存空间，存放的是结构变量的内存地址。结构变量代表一片内存空间的抽象，与 int 等直接变量类似，所以结构变量作为形参是值传递，结构指针变量作为形参是地址传递。

3. 指针变量指向结构变量

```
struct stu
{
    int num;
    char name[30];
};
struct stu girl;
struct stu *pstu;
pstu=&girl;
```

有了结构指针变量，就能更方便地访问结构变量的各个成员。

结构指针变量访问结构成员语法形式如下：

(*结构指针变量).成员名

或为：

结构指针变量->成员名

此时 (*pstu).num 等价于 pstu->num。

4. 结构指针变量作为形参

p_struct.c 源代码如下：

```
#include <stdio.h>
#include <string.h>
struct student
{
    char name[20];
    int age;
    int sex;
    int height;
};
void showstu(struct student *p)
{
    printf("A student:\n");
    printf("  Name  : %s \n",p->name);
    printf("  Age   : %d \n",p->age);
    printf("  Sex   : %d \n",p->sex);
    printf("  Height: %d \n\n",p->height);
}
```

```
}
void main()
{
    struct student stu1;
    struct student *p1;
    p1=&stu1;
    stu1.age=17;
    stu1.sex=1;
    stu1.height=176;
    strcpy(stu1.name,"Jim");
    showstu(p1);
}
```

编译 gcc p_struct.c -o p_struct。

执行 ./p_struct，执行结果如下：

```
A student:
Name  : Jim
Age   : 17
Sex   : 1
Height: 176
```

6.3.11 动态存储分配

变量空间是在栈上分配，动态存储分配是在堆上分配。动态内存分配需要用 free 函数释放空间，没有用 free 释放会造成内存泄露。

1. 动态存储函数原型

申请内存空间函数有 malloc、calloc、realloc 三个函数，这里主要介绍 malloc 函数，另外两个函数将在 Linux 进程编程章节中加以介绍，这三个函数申请的内存空间需要用 free 函数来释放。

malloc 函数和 free 函数的函数原型如下：

malloc（动态申请内存空间）	
所需头文件	#include <stdlib.h>
函数说明	动态申请内存空间
函数原型	void *malloc(size_t size)
函数传入值	size: 申请内存空间的大小
函数返回值	成功: 返回申请内存空间的首地址
	失败: NULL

free（释放原先申请内存空间）	
所需头文件	#include <stdlib.h>
函数说明	参数 ptr 为指向先前由 malloc()、calloc()或 realloc()所返回的内存指针，调用 free()后 ptr 所指的内存空间便会被收回
函数原型	void free(void *ptr)
函数传入值	ptr: 申请内存空间的首地址

2. 动态存储函数举例

malloc.c 源代码如下:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    struct stu
    {
        int num;
        char *name;
        char sex;
        float score;
    } *ps;
    ps=(struct stu*)malloc(sizeof(struct stu));
    ps->num=102;
    ps->name="Zhang ping";
    ps->sex='M';
    ps->score=62.5;
    printf("Number=%d\nName=%s\n",ps->num,ps->name);
    printf("Sex=%c\nScore=%.2f\n",ps->sex,ps->score);
    free(ps);
    return 0 ;
}
```

编译 gcc malloc.c -o malloc。

执行 ./malloc, 执行结果如下:

```
Number=102
Name=Zhang ping
Sex=M
Score=62.50
```

6.3.12 指针链表

链表是通过系统不断申请内存, 然后通过结构体中的指针变量将所申请的空间一级一级链接起来。

以学生链表为例, 下面是链表的结构体定义。

```
struct stu
{
    int num;
    int score;
    struct stu *next;
}
```

图 6-11 为一简单链表的示意图, 通过指针变量把各节点链在一起。

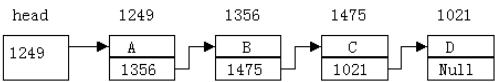


图 6-11 指针链表图

图 6-11 中，第 0 个节点称为头节点，它存放第一个节点的首地址，它没有数据，只是一个指针变量。以下的每个节点都分为两个域，一个是数据域，存放各种实际的数据，如成绩 score 等；另一个域为指针域，存放下一节点的首地址。链表中的每一个节点都是同一种结构类型。

对链表的主要操作有以下几种：

- ① 建立链表；
- ② 结构体的查找与输出；
- ③ 插入一个节点；
- ④ 删除一个节点。

【例 6-13】 建立一个三个节点的链表，存放学生数据，假定学生数据结构中只有学号和年龄两项，可编写一个建立链表的函数 creat。

creat.c 源代码如下：

```
#define NULL 0
#define TYPE struct stu
#define LEN sizeof (struct stu)
struct stu
{
    int num;
    int age;
    struct stu *next;
};
TYPE *creat(int n)
{
    struct stu *head,*pf,*pb;
    int i;
    for(i=0;i<n;i++)
    {
        pb=(TYPE*) malloc(LEN);
        printf("input Number and Age\n");
        scanf("%d%d",&pb->num,&pb->age);
        if(i==0)
            pf=head=pb;
        else pf->next=pb;
        pb->next=NULL;
        pf=pb;
    }
    return(head);
}
```

6.3.13 指针数据类型小结

表 6-8 列出了指针数据类型，并对其含义进行了说明。

表 6-8 指针数据类型小结表

定 义	含 义
int i	定义整型变量 i
int *p	p 为指向整型数据的指针变量
int a[n]	定义整型数组 a，它有 n 个元素
int *p[n]	定义指针数组 p，它由 n 个指向整型数据的指针元素组成
int (*p)[n]	p 为指向含 n 个元素的一维数组的指针变量
int f()	f 为带回整型函数值的函数
int *p()	p 为带回一个指针的函数，该指针指向整型数据
int (*p)()	p 为指向函数的指针，该函数返回一个整型值
int **p	p 是一个指针变量，它指向一个指向整型数据的指针变量

第 7 章

C语言预处理

所谓预处理是指在进行编译的第一遍扫描(词法扫描和语法分析)之前所做的工作。预处理是 C 语言的一项重要功能,它由预处理程序负责完成。当对一个源文件进行编译时,系统将自动引用预处理程序对源程序中的预处理部分做处理,处理完毕自动进入对源程序的编译。

7.1 define 宏定义

C 语言提供了多种预处理功能,如宏定义、文件包含、条件编译等。

在 C 语言源程序中允许用一个标识符来表示一个字符串,称为“宏”。被定义为“宏”的标识符称为“宏名”。在编译预处理时,对程序中所有出现的“宏名”,都用宏定义中的字符串去代换,这称为“宏代换”或“宏展开”。

宏定义是由源程序中的宏定义命令完成的。宏代换是由预处理程序自动完成的。在 C 语言中,“宏”分为有参数和无参数两种。

C 语言对宏定义用 `#define` 命令来实现,注意宏定义后字符串需要用圆括号括起来,防止宏代换时产生歧义和隐含错误。

1. 无参宏定义

无参宏定义的一般形式如下:

```
#define 标识符 字符串
```

如 `#define M (y*y+3*y)`,可用 `#undef M` 解除定义。

2. 有参宏定义

有参宏定义的一般形式如下:

```
#define 宏名(形参表) 字符串
```

上述宏定义在字符串中含有各个形参。

有参宏调用的一般形式如下：

宏名(实参表)；

例如：

```
#define M(y) y*y+3*y      /*宏定义*/
.....
k=M(5);                  /*宏调用*/
.....
```

在宏调用时，用实参 5 去代替形参 y，经预处理宏展开后的语句为：

```
k=5*5+3*5
```

7.2 typedef 重定义

typedef 为 C 语言的关键字，作用是作为一种数据类型定义一个新名字。这里的数据类型包括内部数据类型（int、char 等）和自定义的数据类型（struct 等）。

在编程中使用 typedef 的目的一般有两个，一个是给变量一个易记且意义明确的新名字，另一个是简化一些比较复杂的类型声明。

1. 给已知数据类型 long 起个新名字，叫 byte_4，定义方式如下：

```
typedef long byte_4;
```

2. 结构体重定义

结构体重定义有下面两种方式：

```
struct tagMyStruct
{
    int iNum;
    long lLength;
};
typedef struct tagMyStruct MyStruct;
```

这里 MyStruct 实际上相当于 struct tagMyStruct，可以使用 MyStruct varName 来定义变量。

```
typedef struct{
    char plat_no[3+1];
    char plat_name[60+1];
} T_PLAT_PARA;
T_PLAT_PARA s_para ;
```

可以用 T_PLAT_PARA s_para 来定义变量 s_para。

7.3 inline 关键字

1. inline 关键字说明

inline 为把函数替换为函数展开语句，展开在汇编阶段开始。函数的展开是由编译器决定的，这一点对程序员而言是透明的。只有在代码很短的情况下，函数才会被展开，递归调用函数不会被展开。如果过度地使用 inline 关键字，编译器将不会展开函数，以防止代码体积的恶性膨胀。

2. inline 关键字举例

假设有如下内联函数：

```
inline int add(int a, int b)
{
    return a+b ;
}
int c ;
```

当调用 `c=add(1,9)` 时，函数语句展开，变为 `c=1+9`。

7.4 条件编译

1. 条件编译的三种方式

预处理程序提供了条件编译的功能，可以按不同条件去编译不同的程序部分，因而产生不同的目标代码文件，这对于程序的移植和调试是很有用的。

条件编译有三种形式，具体说明如下。

（1）第一种形式

```
#ifdef 标识符
    程序段 1
#else
    程序段 2
#endif
```

它的功能是，如果标识符已被 `#define` 命令定义过，则对程序段 1 进行编译，否则对程序段 2 进行编译。如果没有程序段 2（它为 `空`），格式中的 `#else` 可以没有，即可以写为：

```
#ifdef 标识符
    程序段
#endif
```

用“`#define 标识符`”来表明对标识符进行了定义，如“`#define xxx`”表明对标识符 `xxx` 进行了定义。

(2) 第二种形式

```
#ifndef 标识符
    程序段 1
#else
    程序段 2
#endif
```

与第一种形式的区别是将“`ifdef`”改为“`ifndef`”。它的功能是，如果标识符未被`#define`命令定义过，则对程序段 1 进行编译，否则对程序段 2 进行编译。这与第一种形式的功能正好相反。

如果要防止某标识符被重复定义，实现方法如下：

```
#ifndef INADDR_NONE
#define INADDR_NONE    0xffffffff
#endif
```

(3) 第三种形式

```
#if 常量表达式
    程序段 1
#else
    程序段 2
#endif
```

它的功能是，如常量表达式的值为真（非 0），则对程序段 1 进行编译，否则对程序段 2 进行编译。

2. `#if` 与 `if` 的区别

`#if` 是在编译时起作用，进行条件编译，而 `if` 是在程序运行时起作用，进行条件判断。

3. `#if` 可经常使用的场合

我在项目中，经常使用“`#if 0`”注释掉程序中一段语句，用“`#if 1`”打开程序中一段语句，这比“`/* */`”注释更清晰、更好使用，因为“`/* */`”不能嵌套。使用方法如下：

```
#if 0
    程序段
#endif
```

7.5 头文件的使用

1. 防止头文件重复包含

假设头文件名为 `somefile.h`，可在头文件名前后加 `__`，并将头文件名大写，然后按如下方式定义防止出现头文件重复包含问题。

```
#ifndef __SOMEFILE_H__
#define __SOMEFILE_H__
```

```
... .. // 声明、定义语句
#endif
```

2. 头文件的使用建议及说明

系统头文件用<>符号进行包含，自定义头文件用" "符号进行包含。

建议将全局函数和类型定义集中于一个头文件中，方便程序引用；将常用的系统头文件集中在一个自定义的头文件中，方便程序引用。

没有正确包含库函数头文件，有时编译会不通过；有时编译通过，但会产生警告信息；有时编译通过连警告信息都不产生。没有正确包含头文件时，少量时候代码执行时会产生莫名其妙的错误。

第 8 章

格式化I/O函数

格式化 I/O 函数分为输出函数和输入函数两大类，输入和输出格式是编程应该掌握的细节，同时也是编程时经常需要使用到的知识。

8.1 格式化输出函数

8.1.1 输出函数原型

格式化 I/O 输出函数原型如下：

```
#include <stdio.h>
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);

#include <stdarg.h>
int vprintf(const char *format, va_list ap);
int vfprintf(FILE *stream, const char *format, va_list ap);
int vsprintf(char *str, const char *format, va_list ap);
int vsnprintf(char *str, size_t size, const char *format, va_list ap);
```

具体说明如下：

(1) 返回值：上述函数成功返回格式化输出的字节数（不包括字符串的结尾'\0'），出错返回一个负值，错误原因存在于 `error` 中。

(2) `printf` 函数会把格式化字符串打印到标准输出。

(3) `fprintf` 函数会把格式化字符串输出到指定的文件 `stream` 中。

(4) `sprintf` 函数会把格式化字符串输出到缓冲区 `str` 中，并在末尾加'\0'，当 `str` 空间不够时，会造成缓冲区溢出。

(5) `snprintf` 函数会把格式化字符串输出到缓冲区 `str` 中，并在末尾加'\0'，但格式化

字符串长度超过 `size-1` 字节时，对格式化字符串按长度 `size-1` 进行截断，因此 `snprintf` 函数比 `sprintf` 函数使用起来更加安全。

（6）上面列出的后四个函数在前四个函数名的前面多了个“`v`”，表示可变参数，不是以...的形式传进来，而是以 `va_list` 类型传进来。

8.1.2 输出函数格式说明

1. format 格式说明

```
%(flags)(width)(.prec)type
```

以上圆括号括起来的参数为选择性参数，而%与 `type` 则是必要的。下面是 `format` 各参数的详细说明。

type 选项说明	
选 项	说 明
整数	
d	整数的参数会被转换成一有符号的十进制数字
u	整数的参数会被转换成一无符号的十进制数字
o	整数的参数会被转换成一无符号的八进制数字
x	整数的参数会被转换成一无符号的十六进制数字，并以小写 <code>abcdef</code> 表示
X	整数的参数会被转换成一无符号的十六进制数字，并以大写 <code>ABCDEF</code> 表示浮点型数
f	<code>double</code> 型的参数会被转换成十进制数字，并取到小数点以下六位，四舍五入
e	<code>double</code> 型的参数以指数形式打印，有一个数字会在小数点前，六位数字在小数点后，而在指数部分会以小写的 <code>e</code> 来表示
E	与 <code>%e</code> 作用相同，唯一区别是指数部分将以大写的 <code>E</code> 来表示
g	<code>double</code> 型的参数会自动选择以 <code>%f</code> 或 <code>%e</code> 的格式来打印，其标准是根据欲打印的数值及所设置的有效位数来决定
G	与 <code>%g</code> 作用相同，唯一区别在以指数形态打印时，会选择 <code>%E</code> 格式
字符串	
c	整型数的参数会被转换成 <code>unsigned char</code> 型打印出
s	指向字符串的参数会被逐字输出，直到出现 <code>NULL</code> 字符为止
p	如果参数是“ <code>void *</code> ”型指针，则使用十六进制格式显示

pre 选项说明
① 正整数的最小位数
② 在浮点型数中代表小数位数
③ 在 <code>%g</code> 格式代表有效位数的最大值
④ 在 <code>%s</code> 格式代表字符串的最大长度
⑤ 若为 <code>x</code> 符号则代表下个参数值为最大长度

width 选项说明
<code>width</code> 为参数的最小长度，若此栏并非数值，而是 <code>*</code> 符号，则表示以下一个参数作为参数长度

flag 选项说明	
选 项	说 明
#	此旗标会根据其后转换字符的不同而有不同含义。当在类型为 o 之前（如%#o），则会在打印八进制数值前多打印一个 o。而在类型为 x 之前（%#x），则会在打印十六进制数前多打印'0x'，在型态为 e、E、f、g 或 G 之前则会强迫数值打印小数点。在类型为 g 或 G 之前时，则同时保留小数点及小数位数末尾的零
u	一般在打印负数时，printf()会加印一个负号，整数则不加任何负号。此旗标会使得在打印正数前多一个正号(+)
o	整数的参数会被转换成一无符号的八进制数字
0	当有指定参数时，无数字的参数将补上 0。默认是关闭此标记，所以一般会打印出空白字符
-	格式化后的内容居左，右边可以留空格

2. 输出字符串中字符类型说明

参数 format 字符串可包含下列 3 种字符类型：

- ① 一般文本，伴随直接输出。
- ② ASCII 控制字符，如\t、\n 等。
- ③ 格式转换字符。

格式转换由一个百分比符号（%）及其后的格式字符所组成。一般而言，每个%符号在其后都必须有一个参数与之相呼应（只有当%%转换字符出现时，会直接输出%字符）。

3. 常用格式化输出说明

格式化输出常用格式为：`%(+|-|0)m.n`。

对格式化输出常用格式解释如下：

- ① m：输出数据域宽，数据长度<m，左补空格，否则按实际输出。
- ② .n：对实数，指定小数点后位数（四舍五入）；对字符串，指定字符串实际输出位数，超过指定长度则进行截断。
- ③ -：输出数据在域内左对齐（默认右对齐）。
- ④ +：指定在有符号数的正数前显示正号（+）。
- ⑤ 0：输出数值时，指定左边不使用的空位置自动填 0。
- ⑥ 输出百分号需要用“%%”。

4. 常用格式化输出使用举例

在实际应用编程中，常用的格式化输出有下面 6 种。

- ① 数字前补 0：`sprintf(acStr,"%06dSECCTL ",30)`。

- ② 左对齐: `sprintf(acStr,"%-6.6s","05023")`。
- ③ 右对齐: `sprintf(acStr,"%6.6s","05023")`。
- ④ 两位小数点输入: `sprintf(acStr,"%2f",19.79)`。
- ⑤ 增加输出百分号: `sprintf(acStr,"%%.2f",19.79)`。
- ⑥ 指定最长输出位数: `sprintf(acStr,"%6.6s","testTEST")`。

5. 输出函数应用举例

`printf.c` 代码如下:

```
#include <stdio.h>
int main()
{
    int i = 150;
    int j = -100;
    double k = 3.14159;
    printf("%d %f %x\n",j,k,i);
    printf("%010d|%-6d|%4d|*d\n",i,i,i,2,i); /*参数 2 会代入格式*中, 而与%2d 同意义*/
    return 0 ;
}
```

编译 `gcc printf.c -o printf`。

执行 `./printf`, 执行结果如下:

```
-100 3.141590 96
0000000150|150   | 150|150
```

变参使用范例 (了解即可)。

`vprintf.c` 代码如下:

```
#include <stdio.h>
#include <stdarg.h>
int my_printf( const char *format,...)
{
    va_list ap;
    int retval;
    va_start(ap,format);
    printf("my_printf( ):");
    retval = vprintf(format,ap);
    va_end(ap);
    return retval;
}
int main()
{
    int i = 150,j = -100;
    double k = 3.14159;
    my_printf("%d %f %x\n",j,k,i);
    my_printf("%2d %*d\n",i,2,i);
}
```

```
    return 0 ;  
}
```

编译 `gcc vprintf.c -o vprintf`。

执行 `./vprintf`，执行结果如下：

```
my_printf( ):-100 3.141590 96  
my_printf( ):150 150
```

8.2 格式化输入函数

8.2.1 输入函数原型

格式化 I/O 输入函数原型如下：

```
#include <stdio.h>  
int scanf(const char *format, ...);  
int fscanf(FILE *stream, const char *format, ...);  
int sscanf(const char *str, const char *format, ...);
```

具体说明如下：

(1) 返回值：上述函数返回成功匹配和赋值的参数个数，成功匹配的参数可能少于所提供的赋值参数，返回 0 表示一个都不匹配，出错返回 -1 并设置 `errno`。

(2) `scanf` 函数会将输入的数据根据参数 `format` 字符串来进行转换并格式化数据。

(3) `fscanf` 函数从指定的文件 `stream` 中读字符。

(4) `sscanf` 函数从指定的字符串 `str` 中读字符。

8.2.2 输入函数格式说明

1. format 类型

`%[*][size][l][h]type`

以上圆括号括起来的参数为选择性参数，而 `%` 与 `type` 则是必要的，具体说明如下（`type` 选项说明如表 8-1 所示）：

- `*` 代表该对应的参数数据忽略不保存。
- `size` 为允许参数输入的数据长度。
- `l`：输入的数据数值以 `long int` 或 `double` 型保存。
- `h`：输入的数据数值以 `short int` 型保存。

表 8-1 type 选项说明

选 项	说 明
d	输入的数据会被转换成一有符号的十进制数字（int）
i	输入的数据会被转换成一有符号的十进制数字，若输入数据以“0x”或“0X”开头，代表转换十六进制数字，若以“0”开头则转换八进制数字，其他情况代表十进制
o	输入的数据会被转换成一无符号的八进制数字
u	输入的数据会被转换成一无符号的正整数
x	输入的数据为无符号的十六进制数字，转换后存于 unsigned int 型变量中
X	同%x
f	输入的数据为有符号的浮点型数，转换后存于 float 型变量中
e	同%f
E	同%f
g	同%f
s	输入数据为以空格字符为终止的字符串
c	输入数据为单一字符
[]	读取数据但只允许括号内的字符，如[a-z]
[^]	读取数据但不允许中括号的^符号后的字符出现，如[^0-9]

2. 格式化输入字符串种类

格式化输入字符串中字符类型包括如下 3 种：

- ① 空格或 Tab，在处理过程中被忽略。
- ② 普通字符（不包括%）和输入字符中的非空白字符相匹配，输入字符中的空白字符是指空格、Tab、\r、\n、\v、\f。
- ③ 格式转换是以%开头，以转换字符结尾，中间有若干个可选项。

3. scanf 函数特别说明

scanf 函数使用格式为：scanf("格式控制串", 地址表)。输入变量默认间隔是空格，如果用“,”号做间隔，输入时也要输入“,”号。

4. 输入函数应用举例

scanf.c 源代码如下：

```
#include <stdio.h>
int main()
{
    int i;
    unsigned int j;
    char s[5];
    scanf("%d %x %5[a-z] %*s %f",&i,&j,s,s);
    printf("%d %d %s\n",i,j,s);
    return 0 ;
}
```

编译 `gcc scanf.c -o scanf`。

执行 `./scanf`，执行结果如下：

```
输入： 10 0x1b aaaaaaaaaa bbbbbbbbbbb
10 27 aaaaa
```

5. 变参实用项目实例

下面是一个打印源代码文件名称、报错位置和信息实用日志函数，具体说明如下：

(1) 程序中 `syslog` 函数打印信息长度不限，报错参数个数和信息不限，并打印该进程进程号。

(2) 日志位置存放在自定义的目录下，日志文件名称可自定义。

(3) 日志文件按天生成，打印函数自动检测日志文件是否存在，如不存在建立相应的日志文件。

(4) 变参一般都需使用 `va_start` 和 `va_end` 函数，编程模式类似，读者可模仿掌握。

`syslog.c` 源代码如下：

```
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdarg.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
/*-----
 * Function Name : syslog
 * Description   : 写日志
 * Input         : sysname    -- 系统代号
 *               : modname    -- 功能模块名
 *               : file_name   -- 文件名
 *               : line_num    -- 行号
 *               : format      -- 消息
 * Output        :
 * Return        : success:0
 *               : fail       :-1
 *-----*/
int syslog( char *sysname, char *modname, char *file_name, int line_num, char
*format, ... )
{
    va_list ap ;
    struct tm *ts;
    time_t now;
    char acTimeStr[16] ;
    char filename[128] ;
    char syscmd[256] ;
    FILE *LogFileID ;
    memset( filename, 0x00, sizeof( filename ) );
    now = time(NULL);
```

```

ts = localtime( &now );
sprintf( filename, "%s/log/%s/%s.%02d%02d.log",
        getenv("HOME"), sysname, modname, ts->tm_mon+1, ts->tm_mday );
LogFileID=fopen( filename, "a" );
if ( LogFileID == NULL )
{
    sprintf( syscmd, "/bin/mkdir -p %s/log/%s/", getenv("HOME"), sysname );
    if ( system( syscmd ) != 0 )
    {
        fprintf(stderr,"creat log folder fail [%s]", syscmd );
        return -1;
    }
    LogFileID=fopen( filename, "a" );
    if ( LogFileID == NULL )
    {
        fprintf(stderr,"open file fail [%s]", filename );
        return -1 ;
    }
}
sprintf( acTimeStr, "%02d-%02d %02d:%02d:%02d",
        ts->tm_mon+1, ts->tm_mday, ts->tm_hour, ts->tm_min, ts->tm_sec);
fprintf( LogFileID, "[%s %d %s:%d] ", acTimeStr, getpid(), file_name,
line_num );
va_start(ap, format );
vfprintf( LogFileID, format, ap );
va_end( ap );
fputc( 0x0a, LogFileID );
fflush( LogFileID );
fclose( LogFileID );
return 0 ;
}
int main()
{
    syslog( "newSys", "SOCK", __FILE__, __LINE__, "hello world! %s %d", "what who
why where when how", 999 );
    syslog( "newSys", "SOCK", __FILE__, __LINE__, "every day is new day!");
    return 0;
}

```

编译 gcc syslog.c -o syslog。

执行 ./syslog, 在\$HOME/log/newSys 下产生了 SOCK.0116.log 文件, 文件内容如下:

```

[01-16 19:51:04 11921 syslog.c:66] hello world! what who why where when how 999
[01-16 19:51:04 11921 syslog.c:67] every day is new day!

```

第 9 章

字符串和内存操作函数

9.1 字符串操作函数说明

对一串字符的处理在应用编程中无处不在，其操作函数主要有两类：一类是以 `str` 开头的函数，主要针对字符串进行处理；一类是以 `mem` 开头的函数，针对一片内存进行处理，此类函数可以处理字符串和结构体。

字符串操作函数总结说明如下。

1. `str` 与 `mem` 的含义

`str` 开头的函数为字符串函数，字符串函数遇到 `'\0'` 结束符即终止操作，字符串最后一个字符即为 `'\0'` 字符。

`mem` 开头的函数为一片内存操作函数，函数原型中包括处理长度，表示对一片内存空间的处理，这片内存中可以包含 `'\0'` 字符。

2. 会对目的字符串自动补 `'\0'` 字符的函数

常见有 `strcpy`、`sprintf`、`gets`、`strcat` 等不指定长度的字符串函数。

3. `strcpy` 与 `memcpy` 的比较

对 `strcpy`、`memcpy` 两函数，差别说明如下：

① `strcpy` 用于字符串的复制，`strcpy` 碰到源串 `'\0'` 字符即终止复制。

② `memcpy` 为一片内存空间的复制，复制时可以包括 `'\0'` 字符，一片内存空间或结构体变量之间的复制常用此函数。

③ `strcpy(dest,src)` 表示把源字符串 `src` 复制到目的字符串 `dest` 中，当 `dest` 空间小于 `src` 长度时，会造成内存溢出，从而让程序 `coredump`。

④ `memcpy (dest,src,n)` 表示复制 `src` 所指的内存内容前 `n` 个字节到 `dest` 所指的内存

地址上。如果是字符串复制，可以用 `strncpy` 代替，不过 `strncpy` 遇到 `'\0'` 字符会终止复制，可能复制不到 `n` 个字节。

⑤ 两结构体变量之间的复制只能用 `memcpy` 函数。

4. `strcpy` 与 `strncpy` 的比较

对 `strcpy`、`strncpy` 两函数，差别说明如下：

① 用 `strcpy` 函数复制时，目的字符串后会自动补 0，用 `strncpy` 函数复制时，目的字符串后不会自动补 0。

② `strncpy(dest,src,n)` 表示从 `src` 源字符串复制 `n` 个字节到 `dest` 目的字符串，但碰到 `'\0'` 字符时会提前终止复制。

③ 用 `strncpy` 函数时，有时需要给目的字符串设置结束符 (`'\0'`)，给一串字符设置 `'\0'` 的方式有两种，分别为 `string[n]=0` 或 `string[n]='\0'`。因为字符可以当做整型单独赋值，0 表示 ASCII 编号的 0；也可以当做字符赋值，字符赋值时特殊字符需用 `\` 来转义。字符 `'0'` 的 ASCII 编号为 30，而字符 `'\0'` 的 ASCII 编号为 0。

9.2 字符串函数操作

字符串函数是编程中常用到的函数，字符串函数按照用途可分为字符串初始化、取字符串长度、复制字符串、连接字符串、比较字符串、搜索字符串、分割字符串七大类。

1. 字符串初始化

(1) 函数原型

memset (将一段内存空间填入某值)	
所需头文件	#include <string.h>
函数说明	memset() 会将参数 <code>s</code> 所指的内存区域前 <code>n</code> 个字节以参数 <code>c</code> 填入，然后返回指向 <code>s</code> 的指针。在编写程序时，若需要对某一数组初始化，memset() 会相当方便
函数原型	void * memset (void *s ,int c, size_t n)
函数传入值	<code>s</code> : 字符串地址
	<code>c</code> : 初始化字符
	<code>n</code> : 初始化字符串长度
函数返回值	返回指向 <code>s</code> 的指针
附加说明	参数 <code>c</code> 虽声明为 <code>int</code> ，但必须是 <code>unsigned char</code> ，所以范围为 0~255
使用场合	① 初始化字符串 ② 初始化结构体 ③ 初始化 <code>malloc</code> 申请的内存空间

(2) `memset` 函数说明

`memset` 函数完成内存一片空间的初始化，对此函数使用方法说明如下：

- ① 字符串初始化: `memset(string,0x00,sizeof(string))`, `string` 表示字符数组。
- ② 结构变量初始化: `memset(&struct,0x00,sizeof(&struct))`, `struct` 表示结构变量。因为结构变量表示一片内存空间的抽象, 需要用`&`得到结构变量地址。
- ③ 字符串和结构变量使用前最好用 `memset` 函数进行初始化。

(3) `memset` 应用代码范例

```
memset.c 代码如下:

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main()
{
    char s[30];
    char *p ;
    struct stu
    {
        int num;
        char name[20];
        char sex;
        float score;
    };
    struct stu boy1;
    p=(char *)malloc(1024);
    memset (s, 0x00, sizeof(s));
    memset(p, 0x00, 1024) ;
    memset(&boy1, 0x00, sizeof(struct stu)) ;
    return 0 ;
}
```

2. 取字符串长度

(1) 函数原型

strlen (返回字符串长度)	
所需头文件	#include <string.h>
函数说明	strlen()用来计算指定的字符串 s 的长度, 不包括结束字符'\0'
函数原型	size_t strlen(const char *s)
函数传入值	s: 字符串首地址
函数返回值	返回字符串 s 的字符数

(2) 应用代码范例

```
strlen.c 代码如下:

#include <stdio.h>
#include <string.h>
int main()
{
    char *str = "12345678";
    printf("str length = %d\n", strlen(str));
}
```

```
    return 0 ;
}
```

编译 gcc strlen.c -o strlen。

执行 ./strlen，执行结果如下：

```
str length = 8
```

3. 复制字符串

(1) 函数原型

strcpy（复制字符串）	
所需头文件	#include <string.h>
函数说明	strcpy() 会将参数 src 字符串复制到参数 dest 所指的地址
函数原型	char *strcpy(char *dest,const char *src)
函数传入值	dest: 目的字符串地址
	src: 源字符串地址
函数返回值	返回参数 dest 的字符串起始地址
附加说明	如果参数 dest 所指的内存空间不够大，可能会造成缓冲溢出（buffer Overflow）的错误情况，在编写程序时请特别注意，或者用 strncpy() 来取代

strncpy（根据长度复制字符串）	
所需头文件	#include <string.h>
函数说明	strncpy() 会将参数 src 字符串的前 n 个字符复制到参数 dest 所指的地址
函数原型	char * strncpy(char *dest,const char *src,size_t n)
函数传入值	dest: 目的字符串地址
	src: 源字符串地址
	n: 复制的字符数
函数返回值	返回参数 dest 的字符串起始地址

memcpy（复制内存内容）	
所需头文件	#include <string.h>
函数说明	memcpy() 用来复制 src 所指的内存内容前 n 个字节到 dest 所指的内存地址上。与 strcpy() 不同的是，memcpy() 会完整地复制 n 个字节，不会因为遇到字符串结束'\0' 而结束
函数原型	void * memcpy (void * dest ,const void *src, size_t n)
函数传入值	dest: 目的字符串地址
	src: 源字符串地址
	n: 复制字节数
函数返回值	返回指向 dest 的指针
附加说明	① 指针 src 和 dest 所指的内存区域不可重叠 ② 结构体之间的复制需要用 memcpy，而不能用 strcpy

memmove（复制内存内容）	
所需头文件	#include <string.h>

续表

memmove（复制内存内容）	
函数说明	memmove()与memcpy()一样都是用来复制src所指的内存内容前n个字节到dest所指的地址上。不同的是，当src和dest所指的内存区域重叠时，memmove()仍然可以正确地处理，不过执行效率上会比使用memcpy()略慢些
函数原型	void * memmove(void *dest,const void *src,size_t n)
函数传入值	dest: 目的字符串地址
	src: 源字符串地址
	n: 复制字节数
函数返回值	返回指向dest的指针

(2) 应用代码

strcpy.c 源代码如下:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char a[30]="string(1)";
    char b[]="string(2)";
    printf("before strcpy() :%s\n",a);
    printf("after strcpy() :%s\n",strcpy(a,b));

    strncpy( &a[9], b, 9) ;
    a[18]='\0' ;
    printf("after strncpy():%s\n", a) ;
    return 0 ;
}
```

编译 gcc strcpy.c -o strcpy。

执行 ./strcpy, 执行结果如下:

```
before strcpy() :string(1)
after strcpy() :string(2)
after strncpy():string(2)string(2)
```

memcpy.c 源代码如下:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char a[30];
    char b[30]="string\0string";
    int i;

    memset( a, 0x00, sizeof(a) ) ;
    strcpy(a,b);
    printf("strcpy():", a);
    for(i=0;i<30;i++)
        printf("%c",a[i]);
}
```

```
printf("\n") ;
memcpy(a,b,30);
printf("memcpy():");
for(i=0;i<30;i++)
    printf("%c",a[i]);
printf("\n") ;
return 0 ;
}
```

编译 gcc memcpy.c -o memcpy。

执行 ./memcpy，执行结果如下：

```
strcpy():string
memcpy():stringstring
```

4. 连接字符串

(1) 函数原型

strcat（连接两字符串）	
所需头文件	#include <string.h>
函数说明	strcat()会将参数 src 字符串复制到参数 dest 所指的字符串尾。第一个参数 dest 要有足够的空间来容纳要复制的字符串
函数原型	char *strcat (char *dest,const char *src)
函数传入值	dest: 目的字符串地址
	src: 源字符串地址
函数返回值	返回参数 dest 的字符串起始地址
附加说明	如果参数 dest 所指的内存空间不够大，可能会造成缓冲溢出（buffer Overflow）的错误情况，在编写程序时请特别注意

strncat（根据长度连接两字符串）	
所需头文件	#include <string.h>
函数说明	strncat()会将参数 src 字符串复制 n 个字符到参数 dest 所指的字符串尾。第一个参数 dest 要有足够的空间来容纳要复制的字符串
函数原型	char * strncat(char *dest,const char *src,size_t n)
函数传入值	dest: 目的字符串地址
	src: 源字符串地址
	n: 复制的字符数
函数返回值	返回参数 dest 的字符串起始地址
附加说明	strncat 与 strncpy 不同，系统会对目的字符串结尾处自动补 0

(2) 应用代码范例

strcat.c 源代码如下：

```
#include <stdio.h>
#include <string.h>
int main()
```

```
{
    char a[30]="string(1)";
    char b[]="string(2)";
    printf("before strcat(): %s\n",a);
    printf("after strcat(): %s\n",strcat(a,b));
    printf("after strncat(): %s\n",strncat(a,b,6));
    return 0 ;
}
```

编译 gcc strcat.c -o strcat。

执行 ./strcat，执行结果如下：

```
before strcat(): string(1)
after strcat(): string(1)string(2)
after strncat(): string(1)string(2)string
```

5. 比较字符串

(1) 函数原型

strcmp（比较字符串）	
所需头文件	#include <string.h>
函数说明	strcmp()用来比较参数 s1 和 s2 字符串。字符串大小的比较是以 ASCII 码表上的顺序来决定，此顺序亦为字符的值。strcmp()首先将 s1 第一个字符值减去 s2 第一个字符值，若差值为 0，就再继续比较下一个字符，若差值不为 0，则将差值返回。例如，字符串"Ac"和"ba"比较，则会返回字符'A'（65）和'b'（98）的差值（-33）
函数原型	int strcmp(const char *s1,const char *s2)
函数传入值	s1: 字符串 1
	s2: 字符串 2
函数返回值	若参数 s1 和 s2 字符串相同，则返回 0。s1 若大于 s2 则返回大于 0 的值。s1 若小于 s2 则返回小于 0 的值

strncmp（比较规定长度的字符串）	
所需头文件	#include <string.h>
函数说明	用来将参数 s1 中前 n 个字符和参数 s2 字符串做比较
函数原型	int strncmp(const char *s1,const char *s2, size_t n)
函数传入值	s1: 字符串 1
	s2: 字符串 2
	n: 比较字符串的长度
函数返回值	若参数 s1 和 s2 字符串相同，则返回 0。s1 若大于 s2 则返回大于 0 的值。s1 若小于 s2 则返回小于 0 的值

memcmp（比较内存内容）	
所需头文件	#include <string.h>
函数说明	memcmp()用来比较 s1 和 s2 所指的内存区间前 n 个字符。字符串大小的比较是以 ASCII 码表上的顺序来决定的，这个顺序亦为字符的值。memcmp()首先将 s1 第一个字符值减去 s2 第一个字符的值，若差为 0 则再继续比较下一个字符，若差值不为 0 则将差值返回。例如，字符串"Ac"和"ba"比较则会返回字符'A'（65）和'b'（98）的差值（-33）

续表

memcmp（比较内存内容）	
函数原型	int memcmp (const void *s1,const void *s2,size_t n)
函数传入值	s1: 字符串 1
	s2: 字符串 2
	n: 比较字符串的长度
函数返回值	若参数 s1 和 s2 字符串相同，则返回 0。s1 若大于 s2 则返回大于 0 的值。s1 若小于 s2 则返回小于 0 的值
附加说明	strcmp、strncmp 碰到 0 字符即终止，而 memcmp 比较内容中可以包含 0 字符

忽略大小写比较字符串	
所需头文件	#include <string.h>
函数说明	用来将参数 s1（中前 n 个字符）和参数 s2 字符串做比较
函数原型	int strcasecmp(const char *s1, const char *s2)
	int strncasecmp(const char *s1, const char *s2, size_t n)
函数传入值	s1: 字符串 1
	s2: 字符串 2
	n: 比较字符串的长度
函数返回值	若参数 s1 和 s2 字符串相同则返回 0。s1 若大于 s2 则返回大于 0 的值。s1 若小于 s2 则返回小于 0 的值

（2）应用代码范例

strcmp.c 源代码如下：

```
#include <stdio.h>
#include <string.h>
int main()
{
    char *a="aBcDeF";
    char *b="AbCdEf";
    char *c="aacdef";
    char *d="aBcDeF";

    printf("strcmp(a,b) : %d\n",strcmp(a,b));
    printf("strcmp(a,c) : %d\n",strcmp(a,c));
    printf("strcmp(a,d) : %d\n",strcmp(a,d));
    printf("strncmp(a,d,5 ) : %d\n",strncmp(a,d, 5));
    printf("memcmp(a,d,3) : %d\n",memcmp(a,d,3));
    printf("memcmp(a,b) : %d\n",strcasecmp(a,b));
    printf("memcmp(a,b,4) : %d\n",strncasecmp(a,b,4));

    return 0 ;
}
```

编译 gcc strcmp.c -o strcmp。

执行 ./strcmp，执行结果如下：

```
strcmp(a,b) : 1
```

```
strcmp(a,c) : -1
strcmp(a,d) : 0
strncmp(a,d,5 ) : 0
memcmp(a,d,3) : 0
memcmp(a,b) : 0
memcmp(a,b,4) : 0
```

6. 搜索字符串

(1) 函数原型

strstr（在一字符串中查找指定的字符串）	
所需头文件	#include <string.h>
函数说明	strstr()会从字符串 haystack 中搜寻字符串 needle，并将第一次出现的地址返回
函数原型	char *strstr(const char *haystack, const char *needle)
函数传入值	haystack: 源字符串
	needle: 搜索字符串
函数返回值	返回指定字符串第一次出现的地址，否则返回 0

strchr（查找字符串中第一个出现的指定字符）	
所需头文件	#include <string.h>
函数说明	strchr()用来找出参数 s 字符串中第一个出现的参数 c 字符的地址，然后将该字符出现的地址返回
函数原型	char * strchr (const char *s, int c)
函数传入值	s: 源字符串
	c: 匹配字符
函数返回值	如果找到指定的字符则返回该字符所在地址，否则返回 0

strrchr（查找字符串中最后出现的指定字符）	
所需头文件	#include <string.h>
函数说明	strrchr()用来找出参数 s 字符串中最后一个出现的参数 c 字符的地址，然后将该字符出现的地址返回
函数原型	char * strrchr(const char *s, int c)
函数传入值	s: 源字符串
	c: 匹配字符
函数返回值	如果找到指定的字符则返回该字符所在地址，否则返回 0

(2) 应用代码范例

strstr.c 源代码如下：

```
#include <stdio.h>
#include <string.h>
int main()
{
    char * s="0123456789012345678901234567890";
    char *p;

    p= strstr(s,"901");
    printf("%s\n",p);
```



```
p=strchr(s, '5');
printf("%s\n",p);
p=strrchr(s, '8');
printf("%s\n",p);

return 0 ;
}

编译 gcc strstr.c -o strstr。
```

执行 ./strstr, 执行结果如下:

```
9012345678901234567890
56789012345678901234567890
890
```

7. 分割字符串

(1) 函数原型

strtok (分割字符串)	
所需头文件	#include <string.h>
函数说明	strtok()用来将字符串分割成一个个的片段。参数 s 指向欲分割的字符串, 参数 delim 则为分割字符串, 当 strtok()在参数 s 的字符串中发现参数 delim 字符时, 则会将该字符改为'\0'字符。在第一次调用时, strtok()必须给予参数 s 字符串, 往后的调用则将参数 s 设置成 NULL。每次调用成功则返回下一个分割后的字符串指针
函数原型	char * strtok(char *s,const char *delim)
函数传入值	s: 源字符串
	delim: 分割符
函数返回值	返回下一个分割后的字符串指针, 如果已无从分割, 则返回 NULL

(2) 应用代码范例

strtok.c 源代码如下:

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char str[] = "root:x::0:root:/root:/bin/bash: ";
    char *token;
    token = strtok(str, ":");
    printf("%s\n", token);
    while ( (token = strtok(NULL, ":")) != NULL)
        printf("%s\n", token);
    return 0;
}
```

编译 gcc strtok.c -o strtok。

执行 ./strtok, 执行结果如下:

```
root
```

```
x
0
root
/root
/bin/bash
```

9.3 字符类型测试函数

字符类型测试用来测试单个字符的类型。

(1) 字符类型测试函数原型（头文件：`#include <ctype.h>`）

函数原型	函数说明
<code>int isalnum(int c)</code>	测试字符是否为英文或数字
<code>int isalpha(int c)</code>	测试字符是否为英文字母
<code>int isascii(int c)</code>	测试字符是否为 ASCII 码字符
<code>int iscntrl(int c)</code>	测试字符是否为 ASCII 码的控制字符
<code>int isdigit(int c)</code>	测试字符是否为阿拉伯数字
<code>int isgraph(int c)</code>	测试字符是否为可打印字符（ASCII 码 33~126 之间的字符）
<code>int islower(int c)</code>	测试字符是否为小写字母
<code>int isprint(int c)</code>	测试字符是否为包含空格在内可打印字符（ASCII 码 32~126 之间的字符）
<code>int isspace(int c)</code>	测试字符是否为空格字符
<code>int ispunct(int c)</code>	测试字符是否为标点符号或特殊符号
<code>int isupper(int c)</code>	测试字符是否为大写英文字母
<code>int isxdigit(int c)</code>	测试字符是否为 16 进制数字
返回值	测试正确：返回 TRUE 或大于零的值
	测试错误：NULL (0)

(2) 应用代码范例

`ischar.c` 源代码如下：

```
#include <stdio.h>
#include <ctype.h>
int main()
{
    char str[]="123@#FDsP[e>";
    int i;
    for(i=0;str[i]!='\0';i++)
        if(islower(str[i])) printf("%c is a lower-case character\n",str[i]);
    printf("1=%d\n", isxdigit('1')) ;
    printf("2=%d\n", isxdigit('2')) ;
    printf("v=%d\n", isxdigit('v')) ;
    printf("U=%d\n", isupper('U')) ;
    printf("u=%d\n", isupper('u')) ;

    return 0 ;
}
```

编译 `gcc ischar.c -o ischar`。

执行 `./ischar`，执行结果如下：

```
s is a lower-case character
e is a lower-case character
1=4096
2=4096
v=0
U=256
u=0
```

9.4 字符串转换函数

字符串转换函数完成数字到字符串、字符串到数字的转换功能，字符串转换函数是在应用编程中经常使用到的函数。

(1) 字符串转换函数原型

atof（将字符串转换成浮点型数）	
所需头文件	#include <stdlib.h>
函数说明	atof()会扫描参数 <code>nptr</code> 字符串，跳过前面的空格字符，直到遇上数字或正负符号才开始做转换，而再遇到非数字或字符串结束 (' <code>\0</code> ')时才结束转换，并将结果返回。参数 <code>nptr</code> 字符串可包含正负号、小数点或 <code>E</code> (<code>e</code>) 来表示指数部分，如 <code>123.456</code> 或 <code>123e-2</code>
函数原型	double atof(const char *nptr)
函数传入值	<code>nptr</code> : 浮点型指针
函数返回值	返回转换后的浮点型数

atoi（将字符串转换成整型数）	
所需头文件	#include <stdlib.h>
函数说明	atoi()会扫描参数 <code>nptr</code> 字符串，跳过前面的空格字符，直到遇上数字或正负符号才开始做转换，而再遇到非数字或字符串结束 (' <code>\0</code> ')时才结束转换，并将结果返回
函数原型	int atoi(const char *nptr)
函数传入值	<code>nptr</code> : 短整型指针
函数返回值	返回转换后的整型数

atol（将字符串转换成长整型数）	
所需头文件	#include <stdlib.h>
函数说明	atol()会扫描参数 <code>nptr</code> 字符串，跳过前面的空格字符，直到遇上数字或正负符号才开始做转换，而再遇到非数字或字符串结束 (' <code>\0</code> ')时才结束转换，并将结果返回
函数原型	long atol(const char *nptr)
函数传入值	<code>nptr</code> : 长整型指针
函数返回值	返回转换后的长整型数

整型、长整型、浮点型转换为字符串	
所需头文件	#include <stdio.h>
函数说明	通常可以用 sprintf 做变量类型转换
函数原型	int sprintf(char *str,const char * format,...)
转换例子	整型转换为字符串: sprintf(str, "%d", 30)
	长整型转换为字符串: sprintf(str, "%ld", 300000)
	浮点型转换为字符串: sprintf(str, "%.2f", 90.90)

(2) 应用代码范例

ato.c 源代码如下:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char a[]="-100";
    char aa[]="1000000000";
    char *aaa="-100.23";
    int c;
    long cc ;
    float ccc ;
    char str[30];

    c=atoi(a);
    cc = atol(aa) ;
    ccc = atof(aaa) ;
    printf("c=%d\n",c);
    printf("cc=%ld\n",cc);
    printf("ccc=%.2f\n",ccc);
    sprintf(str,"%d%ld%.2f",c, cc,ccc) ;
    printf("str=%s\n", str) ;

    return 0 ;
}
```

编译 gcc ato.c -o ato。

执行 ./ato, 执行结果如下:

```
c=-100
cc=1000000000
ccc=-100.23
str=-1001000000000-100.23
```

第 10 章

标准I/O文件编程

标准 I/O 又称为带缓存的 I/O，标准 I/O 库是由 ANSI C 标准进行规范和说明的，基本所有的操作系统上都支持此库。标准 I/O 库处理了很多细节，例如，缓存分配、优化长度执行 I/O 等。这样，用户不必担心如何选择使用正确的块长度。标准 I/O 库是在系统调用函数基础上构造的，它便于用户使用，但是如果不够深入地了解库的操作，也会带来一些问题。

1. 流和 FILE 对象

缓冲型文件系统把每个设备都转换成一个逻辑设备，叫做流。所有的流具有相同的行为。流基本上与设备无关。有两种类型的流：文本流和二进制流。

所有的初级 I/O 函数都是针对文件描述符的。当打开一个文件时，即返回一个文件描述符，然后该文件描述符就用于后续的 I/O 操作。而对于标准 I/O 库，它们的操作则是围绕流(stream)进行的。

当用标准 I/O 库打开或创建一个文件时，会使一个流与一个文件相结合。当打开一个流时，标准 I/O 函数 fopen 返回一个指向 FILE 对象的指针。该对象通常是一个结构，它包含了 I/O 库为管理该流所需要的所有信息：用于实际 I/O 的文件描述符、指向流缓存的指针、缓存的长度、当前在缓存中的字符数、出错标志等。

应用程序没有必要检验 FILE 对象。为了引用一个流，需将 FILE 指针作为参数传递给每个标准 I/O 函数。在本书中，我们称指向 FILE 对象的指针（类型为 FILE *）为文件指针。

2. 文件概述

所谓“文件”，是指一组相关数据的有序集合，这个数据集叫做文件，数据集的名字叫做文件名。实际上在前面的各章中我们已经多次使用了文件，如源程序文件、目标文件、可执行文件、库文件（头文件）等。

文件通常是驻留在外部介质（如磁盘等）上的，在使用时才调入内存中。从不同的角度可对文件进行不同的分类。从用户的角度看，文件可分为普通文件和设备文件两种。

普通文件是指驻留在磁盘或其他外部介质上的一个有序数据集，可以是源文件、目标文件、

可执行程序，也可以是一组待输入处理的原始数据，或者是一组输出的结果。对于源文件、目标文件、可执行程序可以称为程序文件，对输入、输出数据可称为数据文件。

设备文件是指与主机相连的各种外部设备，如显示器、打印机、键盘等。在操作系统中，把外部设备也看做是一个文件来进行管理，把它们的输入、输出等同于对磁盘文件的读和写。

通常把显示器定义为标准输出文件，一般情况下，在屏幕上显示有关信息就是向标准输出文件输出，如前面经常使用的 `printf` 函数就是这类输出。

通常把键盘定义为标准输入文件，从键盘上输入就意味着从标准输入文件上输入数据，`scanf` 函数就属于这类输入。

从文件编码的方式来看，文件可分为 ASCII 码文件和二进制码文件两种。ASCII 文件也称为文本文件，这种文件在磁盘中存放时，每个字符对应一个字节，用于存放对应的 ASCII 码。

例如，数 5678 的存储形式为：

ASCII 码：	00110101	00110110	00110111	00111000
	↓	↓	↓	↓
十进制码：	5	6	7	8

共占用 4 个字节。

ASCII 码文件可在屏幕上按字符显示。例如，源程序文件就是 ASCII 文件，由于是按字符显示，因此，人们能读懂其文件内容。

二进制文件是按二进制的编码方式来存放文件的。

例如，数 5678 的存储形式为：

00010110 00101110

只占 2 个字节。二进制文件虽然也可在屏幕上显示，但其内容无法读懂。C 系统在处理这些文件时，并不区分类型，都看成是字符流，按字节进行处理。

输入/输出字符流的开始和结束只由程序控制而不受物理符号（如回车符）的控制，因此也把这种文件称为“流式文件”。

流式文件的操作主要有打开、关闭、读、写、定位等各种操作。

10.1 文件打开方式

1. 打开文件的方式

标准 I/O 库打开文件的方式共有 12 种，表 10-1 列出了文件的打开方式及其意义说明。

表 10-1 文件打开方式表

文件打开方式	意义说明
rt	只读打开一个文本文件，只允许读数据
wt	只写打开或建立一个文本文件，只允许写数据
at	追加打开一个文本文件，并在文件末尾写数据
rb	只读打开一个二进制文件，只允许读数据
wb	只写打开或建立一个二进制文件，只允许写数据
ab	追加打开一个二进制文件，并在文件末尾写数据
rt+	读写打开一个文本文件，允许读和写
wt+	读写打开或建立一个文本文件，允许读写
at+	读写打开一个文本文件，允许读，或在文件末追加数据
rb+	读写打开一个二进制文件，允许读和写
wb+	读写打开或建立一个二进制文件，允许读和写
ab+	读写打开一个二进制文件，允许读，或在文件末追加数据

2. 文件打开方式说明

对于文件的打开方式，有以下几点说明：

文件的打开方式由 r、w、a、t、b、+共 6 个字符组成，各字符的含义如下：

- ① r (read)：读。
- ② w (write)：写。
- ③ a (append)：追加。
- ④ t (text)：文本文件，可省略不写。
- ⑤ b (banary)：二进制文件。
- ⑥ +：读和写。

凡用“r”打开一个文件时，该文件必须已经存在，且只能从该文件读出。

用“w”打开的文件只能向该文件写入。若打开的文件不存在，则以指定的文件名建立该文件，若打开的文件已经存在，则将该文件删去，重建一个新文件。

若要向一个已存在的文件追加新的信息，只能用“a”方式打开文件。但此时该文件必须是存在的，否则将会出错。

在打开一个文件时，如果出错，fopen 将返回一个空指针值 NULL。在程序中可以用这一信息来判别是否完成打开文件的工作，并做相应的处理。

把一个文本文件读入内存时，要将 ASCII 码转换成二进制码，而把文件以文本方式写入磁盘时，又要把二进制码转换成 ASCII 码。因此，文本文件的读写要花费较多的转换时间，而对二进制文件的读写则不存在这种转换。

标准输入文件（键盘）、标准输出文件（显示器）、标准出错输出（显示器）是由系统打开的，可直接使用。

3. 标准 I/O 文件函数分类说明

表 10-2 列出了标准 I/O 文件函数的分类及其简要说明。

表 10-2 标准 I/O 文件函数分类表

文件函数分类	函数名称
打开和关闭文件函数	fopen() 和 fclose()
字符读写函数	fgetc() 和 fputc()
字符串读写函数	fgets() 和 fputs()
数据块读写函数	fread() 和 fwrite()
格式化读写函数	fprintf() 和 fscanf()
取得文件流的读取位置	ftell()
移动文件流的读写位置	fseek()
文件结束检测函数	feof()

10.2 标准 I/O 函数说明及程序范例

1. 打开和关闭文件

(1) 函数原型

fopen（打开文件）	
所需头文件	#include <stdio.h>
函数说明	根据文件路径打开文件
函数原型	FILE * fopen(const char * path, const char * mode)
函数传入值	path: 打开的文件路径及文件名
	mode: 代表流的形态，参见文件处理方式
函数返回值	成功: 指向该流的文件指针就会被返回
	出错: 返回 NULL，并把错误代码存在 errno 中

fdopen（将文件描述符转换为文件指针）	
所需头文件	#include <stdio.h>
函数说明	fdopen() 会将参数 fildes 的文件描述符转换为对应的文件指针后返回
函数原型	FILE * fdopen(int fildes, const char * mode)
函数传入值	fildes: 文件描述符
	mode 代表文件指针的流形态，此形态必须和原先文件描述符读写模式相同。mode 方式参见文件处理方式
函数返回值	成功: 转换成功时，返回指向该流的文件指针
	错误: 返回 NULL，并把错误代码存在 errno 中

freopen（打开文件）	
所需头文件	#include <stdio.h>
函数说明	参数 path 字符串包含欲打开的文件路径及文件名, 参数 mode 请参考 fopen() 说明。参数 stream 为已打开的文件指针。freopen() 会将原 stream 所打开的文件流关闭, 然后打开参数 path 的文件
函数原型	FILE * freopen(const char * path, const char * mode, FILE * stream)
函数传入值	path: 文件路径全称
	Mode: 参见文件处理方式
	stream: 原先打开的文件流
函数返回值	成功: 文件顺利打开后, 指向该流的文件指针就会被返回
	错误: 返回 NULL, 并把错误代码存在 errno 中

fclose（关闭文件）	
所需头文件	#include <stdio.h>
函数说明	fclose() 用来关闭先前 fopen() 打开的文件。此动作会让缓冲区内的数据写入文件中, 并释放系统所提供的文件资源
函数原型	int fclose(FILE * stream)
函数传入值	stream: 文件指针
函数返回值	成功: 0
	出错: 返回 EOF, 并把错误代码存到 errno

fileno（返回文件流所使用的文件描述符）	
所需头文件	#include <stdio.h>
函数说明	fileno() 用来取得参数 stream 指定的文件流所使用的文件描述符
函数原型	int fileno(FILE * stream)
函数传入值	stream: 已打开的文件指针
函数返回值	返回文件描述符

（2）打开和关闭函数举例

fopen.c 源代码如下:

```
#include <stdio.h>
int main()
{
    FILE * fp;
    int fd ;
    fp=fopen("noexist","a+");
    if (fp==NULL)
    {
        perror("fopen error") ;
        return;
    }

    fd = fileno( fp ) ;
    printf("fildes no=%d\n", fd ) ;
    fclose(fp);
}
```

```
    return 0 ;
}
```

编译 gcc fopen.c -o fopen。

执行 ./fopen, 执行结果如下:

```
fildes no=3
```

fdopen.c 源代码如下:

```
#include <stdio.h>
int main()
{
    FILE *fp =fdopen(0,"w+");
    fprintf(fp,"%s\n","hello!");
    fclose(fp);
    return 0 ;
}
```

编译 gcc fdopen.c -o fdopen。

执行 ./fdopen, 执行结果如下:

```
hello!
```

freopen.c 源代码如下:

```
#include <stdio.h>
int main()
{
    FILE * fp;
    fp=fopen("/etc/passwd","r");
    if ( NULL == fp )
    {
        perror("fopen error") ;
        return -1;
    }
    printf("first fildes no=%d\n", fileno(fp)) ;
    fp=freopen("/etc/group","r",fp);
    if ( NULL == fp )
    {
        perror("fopen error") ;
        return -1;
    }
    printf("second fildes no=%d\n", fileno(fp)) ;
    fclose(fp);
    return 0 ;
}
```

编译 gcc freopen.c -o freopen。

执行 ./freopen, 执行结果如下:

```
first fildes no=3
second fildes no=3
```

可见 freopen 是先关闭文件描述符，然后重新打开文件描述符。

2. 以字符串为单位的 I/O 函数

(1) 函数原型

fgets（从文件中读取一字符串）	
所需头文件	#include <stdio.h>
函数说明	用来从参数 stream 所指的文件内读入字符串并存到参数 s 所指的内存空间，直到出现换行字符、读到文件尾或是已读了 size-1 个字符为止，最后会加上 NULL 作为字符串结束
函数原型	char * fgets(char * s, int size, FILE * stream)
函数传入值	s: 读取内容存放字符串地址
	size: 所读的最大长度
	stream: 文件指针
函数返回值	成功: 返回 s 的指针
	出错: 返回 NULL

fputs（将一指定的字符串写入文件内）	
所需头文件	#include <stdio.h>
函数说明	用来将参数 s 所指的字符串写入到参数 stream 所指的文件内
函数原型	int fputs(const char * s, FILE * stream)
函数传入值	s: 读取内容存放字符串地址
	stream: 文件指针
函数返回值	成功: 返回写出的字符个数
	出错: 返回 EOF 则表示有错误发生

(2) 函数举例

fgets.c 源代码如下：

```
#include <stdio.h>
int main()
{
    char s[80];
    fputs(fgets(s,80,stdin),stdout);
    return 0 ;
}
```

编译 gcc fgets.c -o fgets。

执行 ./fgets，执行结果如下：

```
this is a test /*输入*/
this is a test /*输出*/
```

3. 以块为单位的 I/O 函数

(1) 函数原型

fread（从文件流读取数据）	
所需头文件	#include <stdio.h>
函数说明	用来从文件流中读取数据，读取的字符数由参数 size*nmemb 来决定。fread() 会返回实际读取到的 nmemb 数目，如果此值比参数 nmemb 小，则代表可能读到了文件尾或有错误发生，这时必须用 feof() 或 ferror() 来决定发生什么情况
函数原型	size_t fread(void * ptr, size_t size, size_t nmemb, FILE * stream)
函数传入值	ptr: 指向欲存放读取进来的数据空间
	size: 读取单个块长度
	nmemb: 读取的块数
	Stream: 已打开的文件指针
函数返回值	实际读取到的 nmemb 数目

fwrite（将数据写到文件流）	
所需头文件	#include <stdio.h>
函数说明	用来将数据写入文件流中。总共写入的字符数由参数 size*nmemb 来决定。fwrite() 会返回实际写入的 nmemb 数目
函数原型	size_t fwrite(const void * ptr, size_t size, size_t nmemb, FILE * stream)
函数传入值	ptr: 指向欲写入的数据地址
	size: 写入单个块长度
	nmemb: 写入的块数
	stream: 已打开的文件指针
函数返回值	实际写入的 nmemb 数目

(2) 函数举例

fwrite.c 源代码如下：

```
#include <stdio.h>
struct record {
    char name[10];
    int age;
};
int main(void)
{
    struct record array[2] = {"Ken", 24}, {"Knuth", 28};
    FILE *fp = fopen("recfile", "w");
    if (fp == NULL) {
        perror("Open file recfile");
        return -1;
    }
    /*变量在内存中为对齐存放，整型占 4 个字节，所以整型变量要存放在能整除 4 的内存地址上*/
    printf("record length=%d\n", sizeof( struct record ));
    printf("record name =%d, address=%d\n", sizeof(array[0].name),
    &array[0].name) ;
    printf("record age=%d, address=%d\n ", sizeof(array[0].age),
    &array[0].age) ;
    fwrite(array, sizeof(struct record), 2, fp);
    fclose(fp);
}
```

```
    return 0;
}
```

编译 gcc fwrite.c -o fwrite。

执行 ./fwrite，执行结果如下：

```
record length=16
record name =10, address=-1081549780
record age=4, address=-1081549768
```

fread.c 源代码如下：

```
#include <stdio.h>
struct record {
    char name[10];
    int age;
};
int main(void)
{
    struct record array[2];
    FILE *fp = fopen("recfile", "r");
    if (fp == NULL) {
        perror("Open file recfile");
        return -1;
    }
    fread(array, sizeof(struct record), 2, fp);
    printf("Name1: %s\tAge1: %d\n", array[0].name, array[0].age);
    printf("Name2: %s\tAge2: %d\n", array[1].name, array[1].age);
    fclose(fp);
    return 0;
}
```

编译 gcc fread.c -o fread。

执行 ./fread，执行结果如下：

```
Name1: Ken      Age1: 24
Name2: Knuth    Age2: 28
```

4. 以字节为单位的 I/O 函数

(1) 函数原型

fputc（将一指定字符写入文件流中）	
所需头文件	#include <stdio.h>
函数说明	将参数 c 转换为 unsigned char 后写入参数 stream 指定的文件中
函数原型	int fputc(int c, FILE * stream)
函数传入值	c: 写入成功的字符
	stream: 已打开的文件指针
函数返回值	成功: 写入的字符，即参数 c
	错误: 返回 EOF (-1) 则代表读取失败
附加说明	putc 原型同 fputc，其作用也相同，但 putc() 为宏定义，非真正的函数调用

fgetc（将从文件中读取一个字符）	
所需头文件	#include <stdio.h>
函数说明	从参数 stream 所指的文件中读取一个字符。若读到文件尾无数据时便返回 EOF
函数原型	int fgetc (FILE * stream)
函数传入值	stream: 已打开的文件指针
函数返回值	成功: 返回读到的字符
	错误: 返回 EOF (-1) 则代表读取失败
附加说明	getc 原型同 fgetc，其作用也相同，但 getc() 为宏定义，非真正的函数调用

getchar（从标准输入设备内读取一个字符）	
所需头文件	#include <stdio.h>
函数说明	getchar() 用来从标准输入设备中读取一个字符，然后将该字符从 unsigned char 转换成 int 后返回
函数原型	int getchar (void)
函数返回值	getchar() 会返回读取到的字符，若返回 EOF，则表示有错误发生

putchar（将指定的字符写到标准输出设备）	
所需头文件	#include <stdio.h>
函数说明	putchar() 用来将参数 c 字符写到标准输出设备
函数原型	int putchar (int c)
函数返回值	putchar() 会返回输出成功的字符，即参数 c。若返回 EOF，则代表输出失败
附加说明	putchar() 函数，是通过 putc(c, stdout) 的宏定义实现的

(2) 函数举例

fgetc.c 源代码如下：

```
#include <stdio.h>
int main()
{
    FILE *fp;
    FILE *fp1;
    int c;
    fp=fopen("exist.txt","r");
    if ( NULL == fp )
    {
        perror("fopen error") ;
        return -1;
    }
    fp1= fopen("noexist.txt","w");
    if ( NULL == fp1 )
    {
        perror("fopen error") ;
        return -1;
    }
    while((c=fgetc(fp))!=EOF)
    {
        fputc(c, fp1) ;
    }
    fclose(fp);
}
```

```
    fclose(fp1);
    return 0 ;
}
```

编译 gcc fgetc.c -o fgetc。

创建 exist.txt 文件并输入内容，执行 ./fgetc。查看结果，发现产生了 noexist.txt，且两文件内容一样。

5. 操作读写位置的函数

(1) 函数原型

fseek（移动文件流的读写位置）			
所需头文件	#include <stdio.h>		
函数说明	用来移动文件流的读写位置。参数 offset 为根据参数 whence 来移动读写位置的位移数		
函数原型	int fseek(FILE * stream, long offset, int whence)		
函数传入值	stream: 已打开的文件指针		
	offset: 根据参数 whence 来移动读写位置的位移数，可为正、负、0		
	Whence（既可以用宏，也可以用数字）		
	起始点	宏表示符号	数字表示
	文件首	SEEK_SET	0
	当前位置	SEEK_CUR	1
	文件末尾	SEEK_END	2
函数返回值	成功: 0		
	错误: -1, errno 会存放错误代码		
常见使用方法	① 欲将读写位置移动到文件开头时，用 fseek(FILE *stream,0,SEEK_SET) ② 欲将读写位置移动到文件尾时，用 fseek(FILE *stream,0,SEEK_END)		

rewind（重设文件流的读写位置为文件开头）	
所需头文件	#include <stdio.h>
函数说明	rewind() 用来把文件流的读写位置移至文件开头。参数 stream 为已打开的文件指针，此函数相当于调用 fseek(stream,0,SEEK_SET)
函数原型	void rewind(FILE * stream)
函数传入值	stream: 已打开的文件指针

ftell（取得文件流的读取位置）	
所需头文件	#include <stdio.h>
函数说明	ftell() 用来取得文件流目前的读写位置
函数原型	long ftell(FILE * stream)
函数传入值	stream: 已打开的文件指针
函数返回值	成功: 目前的读写位置
	错误: -1, errno 会存放错误代码
错误代码	EBADF: 参数 stream 无效或文件流的读写位置无效

feof（检查文件流是否读到了文件尾）	
所需头文件	#include <stdio.h>
函数说明	用来检测是否读取到了文件尾。如果已到文件尾，则返回非零值，其他情况返回 0
函数原型	int feof(FILE * stream)
函数传入值	stream: 已打开的文件指针
函数返回值	返回非零值代表已到达文件尾

（2）函数举例

fseek.c 源代码如下：

```
#include <stdio.h>
int main()
{
    FILE * stream;
    long offset;
    stream=fopen("/etc/passwd","r");
    fseek(stream,5,SEEK_SET);
    printf("offset=%d\n",ftell(stream));
    rewind(stream);
    printf("offset = %d\n",ftell(stream));
    printf("file eof status=%d\n", feof(stream)) ;
    fseek(stream,0,SEEK_END);
    printf("offset = %d\n",ftell(stream));
    printf("eof=%d\n", fgetc(stream)) ;
    printf("file eof status=%d\n", feof(stream)) ;
    fclose(stream);
    return 0 ;
}
```

编译 gcc fseek.c -o fseek。

执行 ./fseek，结果如下：

```
offset=5
offset = 0
file eof status=0
offset = 2676
eof=-1
file eof status=1
```

ftell.c 源代码如下：

```
#include <stdio.h>
int main()
{
    FILE *fp;
    long flen;
    char data[10000] ;
    if( ( fp = fopen( "/etc/passwd" , "r" ) ) == NULL ){
        perror("fopen error") ;
        return -1 ;
    }
    fseek(fp,0L,SEEK_END); /* 定位到文件末尾 */
```



```
    flen=ftell(fp); /* 得到文件大小 */
    fseek(fp,0L,SEEK_SET); /* 定位到文件末尾 */
    fread(data,flen,1,fp); /* 一次性读取全部文件的内容 */
    fclose(fp);
    printf("passwd file content:\n%s\n", data ) ;
    return 0 ;
}
```

编译 gcc ftell.c -o ftell。

执行 ./ftell，执行结果如下：

```
passwd file content:
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
.....
```

6. 格式化读写函数

(1) 函数原型

fprintf（格式化输出数据至文件）	
所需头文件	#include <stdio.h>
函数说明	fprintf() 会根据参数 format 字符串来转换并格式化数据，然后将结果输出到参数 stream 指定的文件中，直到出现字符串结束（'\0'）为止
函数原型	int fprintf(FILE * stream,const char * format,...)
函数传入值	stream: 文件描述符
	format: 格式项
函数返回值	成功：返回实际输出的字节数
	失败：-1，并把错误代码存于 errno 中

fscanf（格式化字符串输入）	
所需头文件	#include <stdio.h>
函数说明	fscanf() 会从参数 stream 的文件流中读取字符串，再根据参数 format 字符串来转换并格式化数据。格式化转换形式请参考 scanf()。转换后的结构存于对应的参数内
函数原型	int fscanf(FILE * stream,const char *format,...)
函数传入值	stream: 文件描述符
	format: 格式项
函数返回值	成功：返回参数数目
	失败：-1，并把错误代码存于 errno 中

(2) 函数举例

fprintf.c 源代码如下：

```
#include <stdio.h>
int main()
{
    int i = 150;
```

```
int j = -100;
double k = 3.14159;
int fp ;
fp = fprintf(stdout,"%d %f %x \n",j,k,i);
fp = fprintf(stdout,"%2d %*d\n",i,2,i);
printf(" fp=%d\n", fp) ;
return 0 ;
}
```

编译 gcc fprintf.c -o fprintf。

执行 ./fprintf，执行结果如下：

```
-100 3.141590 96
150 150
fp=8
```

fscanf.c 源代码如下：

```
#include<stdio.h>
int main()
{
    int i;
    unsigned int j;
    char s[5];
    int fd ;
    fd =fscanf(stdin,"%d %x %5[a-z] %*s %f",&i,&j,s,s);
    printf("%d %d %s \n",i,j,s);
    printf("fd=%d\n", fd) ;
    return 0 ;
}
```

编译 gcc fscanf.c -o fscanf。

执行 ./fscanf，执行结果如下：

```
10 0x1b aaaaaaaa bbbbbbbbbbb /*从键盘输入*/
10 27 aaaaa
fd=3
```

7. 产生临时文件

mktemp 函数只产生唯一的临时文件名，并不产生临时文件，而且存在安全隐患，不建议使用。mkstemp 函数会产生唯一的临时文件。

(1) 函数原型

mktemp（产生唯一的临时文件名）	
所需头文件	#include <stdlib.h>
函数说明	mktemp() 用来产生唯一的临时文件名，并不创建文件。参数 template 所指的文件名称字符串中最后六个字符必须是 xxxxxx，产生后的文件名会借字符串指针返回
函数原型	char * mktemp(char * template)
函数传入值	template: 所指的文件名称字符串，最后六个字符必须是 xxxxxx

续表

mktemp（产生唯一的临时文件名）	
函数返回值	成功：以指针方式返回产生的文件名
	失败：NULL，并把错误代码存于 errno 中
附加说明	参数 template 所指的文件名称字符串必须声明为数组，如： char template[]= "template-XXXXXX"; 不可用 char * template="template-XXXXXX";

mkstemp（建立唯一的临时文件）	
所需头文件	#include <stdlib.h>
函数说明	mkstemp() 用来建立唯一的临时文件。参数 template 所指的文件名称字符串中最后六个字符必须是 XXXXXX。mkstemp() 会以可读写模式来打开该文件，如果该文件不存在则会建立该文件
函数原型	int mkstemp(char * template)
函数传入值	template：所指的文件名称字符串，最后六个字符必须是 XXXXXX
函数返回值	成功：文件顺利打开后，返回该文件的文件描述符
	失败：如果文件打开失败，则返回 0，并把错误代码存于 errno 中
附加说明	参数 template 所指的文件名称字符串必须声明为数组，如： char template[]= "template-XXXXXX"; 不可用 char * template="template-XXXXXX";

(2) 函数举例

mktemp.c 源代码如下：

```
#include <stdio.h>
int main()
{
    char template[ ]="template-XXXXXX";
    char *file ;
    file = mktemp(template);
    printf("template=%s\n",template);
    printf("file=%s\n", file);
    return 0 ;
}
```

编译 gcc mktemp.c -o mktemp。

执行 ./mktemp，执行结果如下：

```
template=template-oh2jT8
file=template-oh2jT8
```

mkstemp.c 源代码如下：

```
#include <stdio.h>
int main()
{
    int fd;
    char template[ ]="template-XXXXXX";
    fd=mkstemp(template);
```

```
printf("template = %s\n",template);
printf("fd = %d\n", fd);
close(fd);
return 0 ;
}
```

编译 gcc mkstemp.c -o mkstemp。

执行 ./mkstemp，执行结果如下：

```
template = template-rPGs2W
fd = 3
```

8. 错误检测与清除

(1) 函数原型

ferror（检查文件流是否有错误发生）	
所需头文件	#include <stdio.h>
函数说明	ferror()用来检查参数 stream 所指定的文件流是否发生了错误情况，如有错误发生则返回非 0 值
函数原型	int ferror(FILE *stream)
函数传入值	stream: 已打开的文件指针
函数返回值	如果文件流有错误发生则返回非 0 值

clearerr（清除文件流的错误标记）	
所需头文件	#include <stdio.h>
函数说明	Clearerr()清除参数 stream 指定的文件流所使用的错误标记
函数原型	void clearerr(FILE * stream)
函数传入值	stream: 已打开的文件指针
函数返回值	无

9. 文件编程综合实例

源代码 file.c 如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/*去掉字符串左边指定字符*/
char *LTrimChar( char *pBuf, char cDel )
{
    char *pTmp = pBuf ;
    if ( ( cDel != '\0' ) && ( pTmp != NULL ) )
    {
        while ( *pTmp++ == cDel ) ;
        strcpy( pBuf, pTmp - 1 );
    }
    return pBuf ;
}
/*去掉字符串右边指定字符*/
char *RTrimChar( char *pBuf, char cDel )
```

```

{
    char *pTmp = pBuf;
    if ( ( cDel != '\0' ) && ( pTmp != NULL ) )
    {
        for ( pTmp = pBuf + strlen( pBuf ) -1 ;
              ( *pTmp == cDel ) && ( pTmp >= pBuf ) ; *pTmp-- = '\0' );
    }
    return pBuf ;
}

/*去掉字符串两边指定字符*/
char *AllTrimChar( char *pBuf, char cDel )
{
    return LTrimChar( RTrimChar( pBuf, cDel ) , cDel );
}

int main(int argc, char **argv)
{
    char buff_read[128];
    long line_length ;
    char seri_no[8+1];
    char cust_acct_no[22+1] ;
    char amt[17+1] ; /*原文件金额除以 100 进行转换*/
    char *ptr;
    FILE *fpr; /*读文件指针*/
    FILE *fpw; /*写文件指针*/
    char file_read[64];
    char file_write[64];
    sprintf( file_read,"%s", "read.txt" );
    if( ( fpr = fopen( file_read,"r" ) ) == NULL ) {
        perror("fopen error\n");
        return -1;
    }
    strcpy( file_write, "write.txt" );
    fpw = fopen( file_write , "w" );
    if( fpw == NULL ) {
        perror("fopen error\n");
        return -1;
    }
    memset( buff_read , 0x00 , sizeof( buff_read ) );
    line_length= sizeof(buff_read) ;
    while( fgets( buff_read, line_length, fpr ) != NULL ) {
        memset(seri_no, 0x00, sizeof(seri_no)) ;
        memset(cust_acct_no, 0x00, sizeof(cust_acct_no)) ;
        memset(amt, 0x00, sizeof(amt)) ;
        memcpy( seri_no , buff_read+0 , 8 );
        AllTrimChar( seri_no, ' ' ) ; /*去掉两边的空格*/
        memcpy( cust_acct_no , buff_read+8 , 22 );
        AllTrimChar( cust_acct_no, ' ' ) ;
        memcpy( amt , buff_read+30 , 17 );
        AllTrimChar( amt, ' ' ) ;
        fprintf(fpw,"%-8.8s%-22.22s%-17.21f\n", seri_no,cust_acct_no, atof(amt)/
100 ) ;
    }
    fclose( fpr );
    fclose( fpw );
}

```

```
    return 0 ;  
}
```

编译 `gcc file.c -o file`。

`read.txt` 文件内容为：

```
123456 12345678909876543210 1234567  
123457 12345678909876543230 1234565
```

执行 `./file`。发现建立了 `write.txt` 文件，其文件内容为：

```
123456 12345678909876543210 12345.67  
123457 12345678909876543230 12345.65
```

第 11 章

Linux C语言开发工具

本章主要介绍 Linux C 语言开发工具，包括编辑工具 vi 与 vim 编辑器、编译工具 gcc、项目组织工具 Makefile 和调试工具 gdb。

11.1 vi 与 vim

11.1.1 vi 与 vim 概述

vi 和 vim 是 UNIX 和 Linux 下的经典编辑器。

vim 是 vi 的增强版本，增加了可视化操作，语法高亮显示。vi 方式打开文件用“vi+文件名”，vim 打开文件用“vim+文件名”。所有 UNIX 和 Linux 默认都安装了 vi，而 vim 一般需要自行安装。

vim 向下兼容 vi 的所有命令。

1. vi 与 vim 命令简要说明

本章节中标示的“***”为重点命令，“**”为常见命令，“*”为一般命令，空为了解命令。

常见参数 w (word) 表示单词、d (delete) 表示删除、s (substitute) 表示字符串替换、p (paste) 表示粘贴、\$ 表示行尾、0 表示行首、^ 表示行首第一个字母 (除空格)。c (change) 表示修改、i (insert) 表示插入、a (append) 表示追加、o (open) 表示插入空行、y (yank) 为复制、r 修改单个字母。% 表示全文、m (mark) 为标记。对 vi (vim) 命令的掌握可通过理解记忆和不断练习实践完成。

2. vi 的操作模式

vi 提供三种操作模式：输入模式 (insert mode)、指令模式 (command mode) 和末行模式。

① 输入模式：在输入模式下，用户可输入文本资料。在输入模式下，按 Esc 键可切换到指令模式下。

② 指令模式：在指令模式下，可进行删除、修改等各种编辑动作。在指令模式下，按输入指令（i、a、o 等）进入输入模式。

③ 末行模式：也称 ex 转义模式。在命令模式下，用户按 “:” 即切换到末行模式。例如，末行模式设置文本行号方法为：set number Enter。

图 11-1 给出了 vi（vim）三种模式的转换流程。

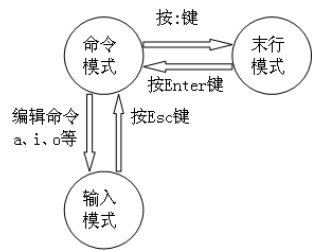


图 11-1 vi（vim）模式转换图

3. 进入 vi

```
$ vi filename [filename filename]
```

直接键入 “vi + 所要编辑的文件名”，即可对指定的文件进行编辑。

如果指定的文件为新文件，则提示：“New file”，否则显示该文件的当前内容。

filename 可以是文件名，也可以是表达式，如*.c、*haha.*。

也可以不指定文件名，直接进入编辑界面，这种方法用于编辑一个新文件，但是暂时还没有确定文件应该叫什么名字时。

```
$ view filenamew 为以只读方式打开文件 filename。
```

表 11-1 列出了 vi 打开文件的方式及其说明。

表 11-1 打开文件方式表

重要度	指令集	功 能
***	vi filename	用 vim 方式打开文件
***	vi *.c	打开当前目录所有*.c 的文件
	vi +6 filename	直接跳到打开文件的第 6 行
	vi + filename	打开文件，并将光标置于最后一行行首
	vi +/main main.c	打开文件直接跳到有 main 的行
	vi -r filename	在上次正用 vi 编辑时发生系统崩溃，恢复 filename
	vi filename1...filenamen	打开多个文件，依次进行编辑。在末行模式敲入 n 命令转入下一个文件
***	view filename	以只读方式打开文件

11.1.2 指令模式

下面所列的所有命令，都必须在指令模式下才能执行。在输入模式下将把输入的字符作为文

件内容添加到文件中（Esc 除外，因为该命令将从输入模式切换到指令模式），而指令模式下输入字符为特定的指令操作，然后进行输入模式。

1. 进入输入模式

表 11-2 列出了 vi 进入输入模式的常规方法。

表 11-2 进入输入模式方法表

重 要 度	指 令 集	功 能
***	a	在当前光标之后输入
	A	在当前行之末输入
***	i	在当前光标之前输入
	I	在当前行之首输入
***	o	在当前行之下新增一行，并在新增行输入
	O	在当前行之上新增一行，并在新增行输入

图 11-2 给出了 I、i、a、A 指令的作用。

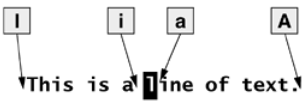


图 11-2 指令说明

表 11-3 列出了切换到输入模式的非常规方法。

表 11-3 非常规模式切换表

重 要 度	指 令 集	功 能
**	s[0 \$...]	替换字符串命令，切换到输入模式，输入的内容将替换指定的内容，直到敲击 Esc 为止
**	c[w 0 \$...]	替换单词命令，切换到输入模式，输入的内容将替换指定的内容，直到敲击 Esc 为止

以上命令在下面有详细介绍，输入结束后，按 Esc 键退出输入模式。

2. 特殊字符

表 11-4 列出了 vi 的特殊字符及其功能说明。

表 11-4 特殊字符表

重 要 度	指 令 集	功 能
***	\	转义字符 碰到 /、\、\$、*、. 时，需要进行转义，在前面需要加 \
	下面是正则表达式	末行模式支持正则表达式
	.	匹配任意单个字符
	.*	匹配多个字符，* 代表匹配多次
	aa.*cc	匹配 aa 开头，cc 结尾的字符串
	[abc]	匹配 a 或 b 或 c

3. 光标移动

表 11-5 列出了 vi 的光标移动方法及其功能说明。

表 11-5 光标移动说明表

重 要 度	指 令 集	功 能
**	h	向左移动一个字节
**	l	向右移动一个字节
*	j	向下移动一个字节
*	k	向上移动一个字节
*	b	左移一个单词，至词首
*	w	右移一个单词，至词首
*	e	右移一个单词，至词尾
***	gg	移动到文档起始位置
***	0（数字零）	移到当前行行首
***	^	移动到本行行首第一个可见字符
***	\$	移到当前行行尾
	+	移至下一行的行首
	-	移至上一行的行首
	H	移至视窗的第一行
	M	移至视窗的中间一行
	L	移至视窗的最后一行
***	G	移至该文件的最后一行
	nG	移至该文档的第 n 行
	N+	下移 n 行
	n-	上移 n 行
	{	光标移动到上一个空行
	}	光标移动到下一个空行
	[[光标移动到下一个函数起始位置（{ 字符所在位置）
]]	光标移动到上一个函数起始位置（{ 字符所在位置）
	(光标移动到上一段落起始位置，空行也被认为是段落（多个连续空行被认为是一个段落）
)	光标移动到下一段落起始位置，空行也被认为是段落（多个连续空行被认为是一个段落）

以上命令从 h 到 e，均可指定重复操作次数，如 5h 表示光标向前移动 5 个字节，10j 表示向下移动 10 行。

图 11-3 形象地给出了 h、k、j、l 指令的光标移动方法图。



图 11-3 h、k、j、l 的光标移动

由于 h、k、j、l 指令一般可以用方向键代替，所以这里把重要度设置成一般，但这影响编辑速度。然而，利用 vi 编辑文档时，有时有的终端不支持方向键，所以这四个指令建议还是要掌

握。正好这四个键在一起，掌握时可优先记住 j 键，j 键的左边有一个 h 键，右边有 k 键和 l 键。h 和 l 与数字结合可实现多个字符右移和左移，如 10h、30l。

4. 视窗移动

表 11-6 列出了 vi 的视窗移动方法及其功能说明。

表 11-6 视窗移动说明表

重 要 度	指 令 集	功 能
***	<Ctrl> + f	视窗下卷一页
***	<Ctrl> + b	视窗上卷一页
	<Ctrl> + d	视窗下卷半页
	<Ctrl> + u	视窗上卷半页
	<Ctrl> + e	视窗下卷一行
	<Ctrl> + y	视窗上卷一行
	zz	将当前行设置为视窗中的中间位置

5. 删除操作

表 11-7 列出了 vi 的删除操作方法及其功能说明。

表 11-7 删除操作说明表

重 要 度	指 令 集	功 能
***	x	删除光标所在字节
	X	删除光标前一字节
**	dw	从光标当前位置开始向后删除一个单词
	db	从光标当前位置开始向前删除一个单词
**	d0	从行首删除至当前光标位置
**	d\$	从光标当前位置删除至行尾
***	dd	删除光标所在行
***	10dd	连续删除 10 行
	D	同 d\$
	dG	删除从当前光标位置直到文档末尾的所有内容

以上 x、X、dw、db、dd 命令前面可以带数字，如 3x 表示删除从当前光标所在位置起的 3 个字符，3dd 表示删除从当前光标所在行开始的 3 行。

6. 复制和粘贴操作

表 11-8 列出了 vi 的复制和粘贴操作方法及其功能说明。

表 11-8 复制和粘贴操作说明表

重 要 度	指 令 集	功 能
**	[n]yy	复制一行或 n 行
**	p	粘贴刚刚复制或者删除内容到当前光标后面，如果是单词则粘贴到当前行，其他内容则在当前光标所在行后面添加新行进行粘贴

7. 修改操作

表 11-9 列出了 vi 的修改操作方法及其功能说明。

表 11-9 修改操作说明表

重 要 度	指 令 集	功 能
***	s	修改光标所在字节，修改完成后按 Esc 键结束
	S	修改整行内容，修改完成后按 Esc 键结束
***	r	替换当前光标所在字节
*	R	进入替换状态，直至按 Esc 键结束
*	cw	修改一个单词（从光标位置至词尾）
	cb	修改一个单词（从词首至光标位置）
*	cc	修改整行内容，完成后按 Esc 键结束，同 S
*	c0	修改行首至光标位置的内容
*	c\$	修改光标位置至行尾的内容

s、r、cw、cb 命令可以指定重复操作次数，如 5s 表示替换当前光标及其以后的 5 个字符，3cb 表示替换当前光标及其之前的 3 个单词。

8. 指令重复

表 11-10 列出了 vi 的指令重复操作方法及其功能说明。

在指令模式中，可在指令前面加入一个数字 n，则该指令会重复执行 n 次。常用的重复操作有表 11-10 中列出的三种。

表 11-10 指令重复说明表

重 要 度	指 令 集	功 能
***	nx	删除 n 个字节
***	ndd	删除 n 行
***	ns	修改 n 个字节

9. 取消前一动作

表 11-11 列出了 vi（vim）取消前一动作的操作方法及其功能说明。

表 11-11 取消前一动作说明表

重 要 度	指 令 集	功 能
***	u	撤销上一指令的结果
**	U	撤销本行上的所有修改

一般 vi 只保存上一次的修改，即本行的所有修改，因此执行 u 指令时，撤销上次修改，再执行 u 指令则撤销“撤销操作”。如果执行了 U，则 u 就没用了。

使用 vim 软件，可以修改软件保存所有的修改，因此 u 可以一直进行撤销，但不能完成撤销“撤销操作”。

10. 查找字符串

表 11-12 列出了 vi 查找字符串的操作方法及其功能说明。

表 11-12 查找字符串说明表

重 要 度	指 令 集	功 能
***	/字符串	从当前光标向后查找该字符串
***	?字符串	从当前光标向前查找该字符串
***	n	从当前光标向后查找下一个字符串
*	N	从当前光标向前查找下一个字符串

11. 查看编辑状况

表 11-13 列出了 vi 查看编辑状况的操作方法及其功能说明。

表 11-13 查看编辑状况说明表

重要度	指令集	功能
***	<Ctrl> + g	显示正在编辑的文件名、当前光标所在行数、文件总行数、文件是否被修改

12. 括号匹配

表 11-14 列出了 vi 查看括号匹配的操作方法及其功能说明。

表 11-14 括号匹配说明表

重要度	指令集	功能
***	%	定位到匹配的“(”、“)”、“{”、“}”

括号匹配符可以用在各种指令中，如 d%表示从当前光标位置删除到匹配的“(”、“)”、“{”、“}”的位置，c%表示替换从当前光标到匹配的“(”、“)”、“{”、“}”位置，y%表示复制从当前光标位置到其后匹配的“) ”或“}”。

13. 标记操作

表 11-15 列出了 vi 标记操作方法及其功能说明。

表 11-15 标记操作说明表

重 要 度	指 令 集	功 能
***	ma （当前行） y’a （移动到其他行） p	块复制 p 把块复制的内容粘贴在当前行下面
***	ma d’a	块删除
	"ema "ey’a :e filename "ep	跨文件复制。 把文件块放在标识 e 的缓冲区中，跳转到另一文件并从缓冲区 e 中取出进行粘贴

说明：m (mark) 为标记，a 为标识缓冲区（可以为其他字母），ma 含义为标记当前 a，复制内容存放在标识 a 的缓冲区中，y 为复制。

14. 重复指令

表 11-16 列出了 vi 重复指令操作方法及其功能说明。

表 11-16 重复指令说明表

重 要 度	指 令 集	功 能
***	.	重复上一条指令

15. 排版命令

表 11-17 列出了 vi (vim) 排版命令操作方法及其功能说明。

表 11-17 排版命令说明表

重 要 度	指 令 集	功 能
***	==	排版当前行（只对 vim 有效）
***	9==	从当前行开始，排版 9 行（只对 vim 有效）
***	gg =G	到第一行，然后进行全文排版（只对 vim 有效）
***	n>>	n 行右移一个 Tab 键
***	n<<	n 行左移一个 Tab 键
	>6 回车	6 行右移一个 Tab 键
	<7 回车	7 行左移一个 Tab 键

16. 大小写切换

表 11-18 列出了 vi 大小写切换命令操作方法及其功能说明。

表 11-18 大小写切换说明表

重 要 度	指 令 集	功 能
*	~	光标处大小写连续切换
	guw	该单词变小写
	gUw	该单词变大写
	gU\$	光标到行末的单词都变成大写

11.1.3 末行模式

1. 保存与退出

表 11-19 列出了 vi 保存与退出操作方法及其功能说明。

表 11-19 保存与退出说明表

重 要 度	指 令 集	功 能
***	:w	保存，但不退出

续表

重 要 度	指 令 集	功 能
***	:wq	保存并退出
	:x	同:wq
***	:q	退出, 如果当前文件没有保存, 禁止退出
***	:q!	不保存退出
	:w file1	将内容保存至文件 file1 中, 此时编辑的仍为原文件, 如文件 file1 存在, :w file1 将禁止执行
	:w! file1	将内容覆盖保存至文件 file1 中。!表示强制执行, 即 file1 存在仍执行

2. 环境设置

表 11-20 列出了 vi 环境设置操作方法, 此时的环境设置只对当前操作环境有效。如果要设置一个用户下 vi 环境变量的默认值, 可在\$HOME/.exrc 文件中设置。

表 11-20 环境设置说明表

重 要 度	指 令 集	功 能
	:set	显示已设置的环境状况
	:set all	显示所有的环境设置选项
*	:set nu	显示行号
*	:set nonu	不显示行号
*	:set ts=n	设置 Tab 键的长度为 n
***	:syntax on	该命令实现语法高亮显示

3. 末行模式执行指令

以下以具体的数字为例进行介绍, 数字表示行数, 光标所在行可用“.”代替, 文件最后一行可用“\$”代替。

条条大路通罗马。vi 达到同一目的有多种实现方式, 指令模式完成的许多功能末行模式也能实现。末行模式 s 替换命令是常使用的命令, 需熟练掌握。

替换命令格式如下:

```
: [range] s/s1/s2/ [option]
```

其中, range 表示范围, 使用方法如下:

- “1,6”表示 1 行~6 行。
- “%”表示整个文件, 同“1,\$”。
- “.,\$”表示从当前行到文件尾。

s 为替换命令, s1 为要被替换的字符串, s2 为替换的字符串。

option 为替换方式, /g 表示在全局文件替换, /c 表示替换前需要用户确认, 为空只替换行首匹配的字符串。

表 11-21 列出了 vi 在末行模式下常用的命令操作及其说明。

表 11-21 末行模式常用命令说明表

重要度	指 令 集	功 能
***	:r filename	读入 filename 文件内容，并粘贴到当前光标下一行
***	:6,60 w a.tmp	将 6 行~60 行写入 a.tmp 文件
***	:s/aaa/ccc	将当前行首次出现的 aaa 替换成 ccc，如 aaaxxxaaazzzaaa 执行后变成 cccxxxaazzzaaa
***	:s/aaa/ccc/g	将当前行的 aaa 全部替换成 ccc，如 aaaxxxaaazzzaaa 执行后变成 cccxxxczzzzccc
***	:9,18 s/aaa/ccc/g	9~18 行的 aaa 全部替换成 ccc
***	:. ,18 s/aaa/ccc/g	当前行到 18 行的 aaa 全部替换成 ccc
***	:9,\$ s/aaa/ccc/g	9 到文件尾的 aaa 全部替换成 ccc
***	:6,16 s/sss\$/ /g	将 6 行~16 行以 sss 结尾的字符以空代替（即删除）
***	:9,19 s/^xxx/yy/g	将 9 行~19 行以行首 xxx 开头的字符以 yy 代替
***	:s/aa/cc/xx/yy/g	将当前行的 aa/cc 替换成 xx/yy
***	:%s/s/aaa/ccc/g	全文将 aaa 全部替换成 ccc，%表示全文
***	:%s/s/aaa/AAA/c	替换前需要确认
***	:6, 19 s/^/AAA	6 行~19 行文件头增加 AAA
***	:6, 19 s/\$/CCC	6 行~19 行文件尾增加 CCC
*	!:pwd	!表示执行外部命令
*	:1,3 !pwd	执行外部命令 pwd 并把结果覆盖文件 1 行~3 行
*	:1, 30 !awk '{print \$1, \$1}'	把文件 1 行~30 行的第一列变成同样的两列，并覆盖原来的 1 行~30 行
*	:g/^\$/d	删除所有空行
***	:help	获取帮助命令
***	:help [key]	显示vi的帮助信息,如果指定了命令,则显示该命令的帮助信息.用:exit 或:q 推出帮助界面
***	:%s/^M//g	由于 UNIX 和 Linux 下文件只有换行符，Windows 下行尾是换行回车。 此操作为去掉回车符 ^M输入方法：同时按下 Ctrl 键和 v 键，释放 v 键按下 M 键，释放 Ctrl 键
..***	:set fileen+tab	利用 Tab 键达到参数补全功能（vim 有效）

表 11-22 列出了 vi 在末行模式下不常见的命令操作及其说明。

表 11-22 了解的命令说明表

重 要 度	指 令 集	功 能
	:10,20d	删除第 10 行~第 20 行的内容
	:10d	删除第 10 行的内容
	:%d	删除全部内容
	:10,20co30	将第 10 行~20 行的内容复制到第 30 行之后
	:10,20mo30	将第 10 行~20 行的内容移动到第 30 行之后
	:10	将光标移至第 10 行
	:10,20y	复制第 10 行~20 行的内容

续表

重 要 度	指 令 集	功 能
	:g/old/s//new	在全文档范围内查找每行第一次出现的 old 字符串并替换为 new 字符串
	:g/old/s//new/g	在全文档范围内查找所有 old 字符串并替换为 new 字符串
	:g/old/d	删除文档中包含 old 字符串的行
	:e	重新载入当前文档的内容覆盖当前所有修改，其实就是将所有自上次保存（或者打开）后的所有修改撤销
	:\$	光标移动到文档末尾一行行首
	:!pwd	在当前行执行外部命令
	:sp(lit)	切割窗口
	:e filename	跳转到其他文件编辑

4. 切换到 Shell 状态

表 11-23 列出了 vi 在末行模式下切换到 Shell 状态的方法。

表 11-23 切换到 Shell 状态说明表

重 要 度	指 令 集	功 能
***	:sh	切换到 Shell 状态，此时可以执行所有 Shell 命令，执行 exit 返回文档编辑状态

11.1.4 vim 个人使用经验

下面是个人常使用的命令，列出来供大家参考。

- ① 熟练掌握 a (append) 表示追加，i (insert) 表示插入，o (open) 表示向下插入空行。
- ② 知道^表示行的第一个字符，0 表示行首，\$表示行末，w (word) 表示单词，数字表示范围，\表示转义字符，gg 表示文件头，G 表示文件尾。
- ③ 知道 d (delete) 表示删除，y (yank) 表示复制，p (paste) 表示粘贴，c (change) 表示修改，v (visual) 表示可视化，=表示排版，.表示重复，m (mark) 表示标记。
- ④ 知道 x 表示删除一个字符，r 表示修改一个字符，s 表示替换一个字符，9s 表示替换 9 个字符，直到按 Esc 键结束替换。
- ⑤ 熟练掌握末行替换命令 s 的用法，知道数字是表示范围和可以使用正则表达式。知道末行模式“!+命令”是执行外部命令，“命令+!”是执行强制命令。
- ⑥ 知道用 w 可以把当前文件写入到另一个文件，用 r 可以读取指定文件内容到当前光标下一行。
- ⑦ 知道用 /、?、n 命令查找特定字符串。
- ⑧ 知道 u 表示撤销，知道可以用标记操作 m (mark) 来完成块复制和块删除。
- ⑨ 知道^[表示 Esc 键，^M 表示回车键。

知道上述含义后，可以使用命令组合进行各种操作。表 11-24 列出了 vi 命令的组合方式，左边命令和右边特殊标识字符可组合完成多种功能。

表 11-24 命令组合表

命令字符	特殊标识字符
d、y、c	^、0、\$、w、数字

表 11-25 列出了 vi 命令组合的使用。

表 11-25 命令组合示例表

命令字符	特殊标识字符	组合指令	含 义
d	10	10dd	删除 10 行
d	w	dw	删除单词
d	^	d^	删除到文件头
d	\$	d\$	删除到行尾
c	w	cw	修改单词
c	w、10	10cw	修改 10 个单词
c	0	c0	修改到文件头
c	\$	c\$	修改到文件尾
y	10	10yy p	复制 10 行然后用 p 粘贴
y	w	yw	复制单词
y	0	y0	复制到行首
y	\$	y\$	复制到行尾
m、y		ma y'a p	块复制
m、d		ma d'a	块删除

11.1.5 vim 的使用

1. vim 可视化命令及宏

表 11-26 列出了 vim 的可视化操作和宏的使用方法。

表 11-26 vim 可视化命令与宏说明表

重 要 度	指 令 集	功 能
***	v	可视化，然后用光标选定范围，选定范围会高亮显示。此时按“:”可以切换到末行模式，末行自动增加“:’<,’>”表示范围。例如： :’<,’> s/aaa/ccc/g 把可视化选定行 aaa 替换成 ccc
*	宏的使用 gg qa /{ dd q 100@a	gg 表示光标移到行首； qa 表示录制宏，/{ 表示光标移到 { 行，dd 表示删除该行，q 表示退出宏录制； 100@a 表示宏 a 执行 100 次

2. vim 配置文件

在当前主目录\$HOME 下配置.vimrc 文件，可设置 vim 文件的默认参数。

下面是一个.vimrc 文件，提供了对 vim 参数的设置方法。

```
set so=3
set history=400
set wildmenu

filetype plugin on
set laststatus=2
set statusline=%-37.37f%-10.10(%m%r%h%)%-11.11(%l:%c%V%)%4P%7L%10y

" tags 文件从当前目录逐层往上找
set tag=tags;/

" 设置开启语法高亮
syntax enable
filetype indent on

" tab 宽度
set tabstop=4
set softtabstop=4
set shiftwidth=4
set expandtab
set listchars=tab:>-,trail:-
set wrap

" 设置文件编码
set fileencodings=cp936,utf8,latin1
" 设置终端编码
set termencoding=utf8
" 设置编码
set encoding=cp936

"colorscheme murphy

map vm O/* */^[2hi
map vM $a /* */^[2hi
map vv ^i/*^[A*/^[j
map tt $?\\/*^M2x/*\\/*^M2xk
map com O/^[55a*^[yypx$pO*^[k3==jla

nmap <silent> <F1> mq:%s/\\s\\+$//^M`q
nmap <silent> <F3> [I
nmap <silent> <F5> =a{``zz

" 在文件中执行 db2 语句。首先用 v 可视化语句，让后用，r 执行语句。zjkf 为数据库名
let mapleader = ","
vmap <leader>r y:bel new^Moconnect to zjkf ^M^[p:%!db2 \\ head -228^Mgg28dd:set
nowrap^M

if &term =~ "xterm" || &term =~ "vt100" || &term =~ "ansi"
    if has("terminfo")
```

```
        set t_Co=8
        set t_Sf=^[[3%p1%dm
        set t_Sb=^[[4%p1%dm
    else
        set t_Co=8
        set t_Sf=^[[3%dm
        set t_Sb=^[[4%dm
    endif
endif
```

3. 与 tags 结合使用

vim 与 tags 结合使用的方法如下：

- ① vim 与 tags 文件结合使用常应用在源程序编辑和查看上。
- ② ctags -R 生成 tags 文件，R 表示递归。
- ③ 在.vimrc 指定 tags 的寻找方式。
- ④ 用 vim -t 函数名，可以直接打开该函数文件。
- ⑤ 打开文件，光标在一个函数上用 Ctrl+] 可以跳转到该函数的定义，用 Ctrl+T 返回。
- ⑥ 如果一个函数多次定义，可以使用 G+] 查看多个函数定义，然后输入数字跳转到指定函数位置。

4. vim 功能键定义

vim 中 map 命令完成功能键宏的定义。

map 命令有许多变化形式，每种变化形式所定义的键只在某些模式下有效，而在其他模式下无效。

需要根据实际情况使用正确的变化形式，具体说明如下。

- :nmap 键只对普通模式有效。
- :imap 键只对插入模式有效。
- :vmap 键只对可视模式有效。
- :cmap 键只在命令行下有效。
- :map 键在普通模式和可视模式下都有效，
- :map! 键在插入模式和命令行下都有效。

应用举例说明：

- “:map m n” 创建一个宏（使 m 做 n）

- “:map! m n” 创建一个插入模式的宏（使 m 做 n）
- “:unmap m” 删除宏 m
- “:unmap! m” 删除插入模式的宏 m

11.1.6 文件编码

1. set 参数含义

末行模式下，设置选项（set）参数语法如下：

```
:set <选项> <参数> （如果需要参数的话）  
显示选项（set）参数语法如下：
```

- :set <选项>? 显示出当前这个选项的参数值
- :set all 显示所有的选项值

对那些不需要的参数选项来说，使用如下语法形式可以关掉：

```
:set no <选项>
```

2. 编码说明

编码：计算机要准确地处理各种字符集文字，需要进行字符编码，如 UTF-8、GB2312、Unicode 编码等。编码只不过是一个字符的代号，同一字在不同编码中代号可能不同。

编码显示：显卡根据字符的编码类型和编码代号将字符转换为点阵像素码，提交给显示器显示。

不同字符集只是记录字符的编号。图 11-4 是“英”字的 16*16 点阵，如“英”字在 x 编码中代号为 1133，在 xx 编码中可能是 6677。x 编码代号 1133 和 xx 编码中的 6677 对照的点阵码都是“英”字。

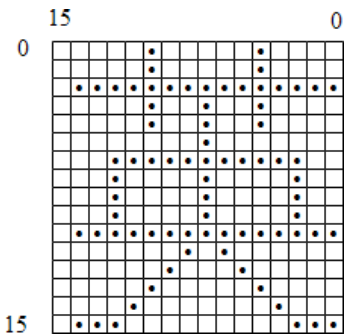


图 11-4 点阵码图

3. 系统 locale 编码

locale 编码为本地语言编码，定义了当前工作环境的默认语言编码。vim 保存在内存中信息的 encoding 编码一般要求与 locale 编码一致。

查看 locale 编码只需要在命令行敲下 locale 即可。

Linux 每个用户可以在 .profile 中设置 locale 编码，如 `LANG=zh_CN.utf8 export LANG`。

4. vim 编码参数

vim 有四个跟字符编码方式有关的选项：encoding、fileencoding、fileencodings、termencoding。它们的意义如下：

① encoding：该选项使用于缓冲的文本（正在编辑的文件）、寄存器、vim 脚本文件等。可以把 encoding 选项当做是对 vim 内部运行机制的设定。

② fileencoding：该选项是 vim 写入文件时采用的编码类型。

③ termencoding：该选项代表输出到客户终端（term）采用的编码类型。

④ fileencodings：vim 启动时会按照它所列出的字符编码方式逐一探测即将打开的文件的字符编码方式，并且将 fileencoding 设置为最终探测到的字符编码方式。因此最好将 Unicode 编码方式放到这个列表的最前面，将拉丁语系编码方式 latin1 放到最后面。

由于文件编码并不保存在文件内。vim 打开文件时是根据 .vimrc 来设定 encoding、termencoding 和 fileencodings 这三个参数的。vim 根据 fileencodings 设置值探测文件类型，从而设置 fileencoding，如 fileencodings 设置为 cp936、UTF-8、latin1，探测到文件类型为 cp936，则 fileencoding 设置为 cp936。

编码变量的默认值：

① encoding：要求与系统当前 locale 相同。

② fileencoding：vim 打开文件时自动辨认的编码，根据 fileencodings 设置进行辨认。fileencodings 为空，则保存文件时采用 encoding 的编码，如果没有设置 encoding，那值就是系统当前的 locale。

③ termencoding：默认为空值，也就是输出到终端不进行编码转换。

5. vim 编码工作原理

vim 编码工作原理如下：

① vim 启动，根据 .vimrc 中设置的 encoding 值来设置 buffer、菜单文本、消息文本的字符编码方式。

② 读取需要编辑的文件，根据 fileencodings 中列出的字符编码方式逐一探测该文件编码方式，并设置 fileencoding 为探测到的、看起来是正确的字符编码方式。

③ 对比 fileencoding 和 encoding 的值，若不同则调用 iconv 将文件内容转换为 encoding 所描述的字符编码方式，并且把转换后的内容放到为此文件开辟的 buffer 里，此时

就可以开始编辑这个文件了。注意，完成这一步动作需要调用外部的 `iconv.dll`。

④ 编辑完成后保存文件时，再次对比 `fileencoding` 和 `encoding` 的值。若不同，再次调用 `iconv` 将即将保存的 `buffer` 中的文本转换为 `fileencoding` 所描述的字符编码方式，并保存到指定的文件中。同样，这需要调用 `iconv.dll`。

6. 几种编码解释

- **locale 编码**：本地语言编码，即默认工作时使用编码。
- **文件编码**：文件保存时所用的编码，即 `fileencoding` 编码。
- **vim 编码**：vim 读取信息时存放在内存中信息的编码，即 `encoding` 编码。
- **终端编码**：终端自身采用的编码。由于早期计算机终端通过外设接口（COM1 串口、RS-232-C 并口）接收字符编码，终端根据字符编码代号找到自己对应的字符点阵码，自己解析并显示。终端类型有许多种，为了终端能正确显示，需要将 `termencoding` 编码参数设置成终端要求的编码。

7. 字符集转换

不同机器上编码文件不一样，可以用 `iconv` 工具进行转换。

字符集转换命令的格式如下：

```
iconv -f 源编码 -t 目标编码 源文件 >目标文件
```

例如，将 `gb1.txt` 里的编码从 GB2312 转化成 UTF-8 并重定向到 `gb2.txt`，如下：

```
iconv -f GB2312 -t UTF-8 gb1.txt >gb2.txt
```

8. 乱码解决思路

在编辑 vim 文件时，终端有时显示为乱码，此时需要从系统 `locale` 编码、终端编码、文件编码、vim 编码四者协调入手，通常要求 `locale` 编码与 vim 编码要一致。对于显示乱码解决思路如下：

① 在 `.vimrc` 中手工设定 `encoding` 编码与 `locale` 编码一致。

② `cat filename` 显示正常，`vim filename` 打开显示不正常。问题在于终端编码设置不对，设置 `termencoding` 值与终端类型编码一致。在 `.vimrc` 中设置“`set termencoding=utf-8`”（终端编码根据具体情况设置，ssh 终端默认接收的数据为 UTF-8）。

③ `cat filename` 显示不正常，说明文件编码与 `locale` 编码不一致。可以在 `.vimrc` 设置“`set fileencodings=cp936,utf8,latin1`”，让 vim 自动识别文件字符编码规则，vim 自动识别后会吧识别结果赋值给 `fileencoding`。

④ 由于 vim 不能百分之百识别文件正确，有时需要在 `.vimrc` 中手工设定 `fileencoding`。

⑤ 如果文件字符编码与当前用户环境 locale 编码不同，可在命令行调用 iconv 命令对文件进行字符集转换，解决乱码问题。

11.1.7 vi 与 vim 模拟练习

编辑文件 vim vim.txt，录入下面的内容。

```
/usr/share/vim/vim71/doc/tags
    The tags file used for finding information in the documentation files.

/usr/share/vim/vim71/syntax/syntax.vim
    System wide syntax initializations.

/usr/share/vim/vim71/syntax/*.vim
    Syntax files for various languages.
```

① 用 dd 删除第三行的空行。

② 用 2yy 复制开头两行，使用 G 将光标移到文件尾，用 p 命令粘贴。

③ 使用 gg 将光标移到文件头，敲入 ma，光标下移四行，敲入 y'a 进行块复制。移动光标到任意一行，使用 p 命令将块复制粘贴。

④ 使用 ma、d'a 将刚才的块复制删除。

⑤ 按下 Esc 键切换到指令模式，用 /tags 找到 tags 单词，使用 cw 命令编辑修改为 TAGS，使用 u 命令恢复回去。

⑥ 使用 \$ 使光标跳到行尾，使用 0 让光标跳到行首，使用 9l 使光标移动 9 格。

⑦ 使用 15x 删除 15 格字符，使用 u 命令恢复回去。

⑧ 使用 d\$ 删除到行尾，使用 u 命令恢复回去。

⑨ 使用 d^ 删除到行首，使用 u 命令恢复回去。

⑩ 使用 yw 复制单词，使用 p 查看粘贴效果，使用 u 命令恢复回去。

⑪ 使用 10yw 复制单词，使用 p 查看粘贴效果，使用 u 命令恢复回去。

⑫ 使用 y^ 复制到行首，使用 p 查看粘贴效果，使用 u 命令恢复回去。

⑬ 使用 y\$ 复制到行尾，使用 p 查看粘贴效果，使用 u 命令恢复回去。

⑭ 使用 o 增加一空行。

⑮ 使用 ?information 查找到单词，使用 i 命令增加字符 why。使用 u 命令恢复回去。

⑯ 使用 ?information 查找到单词，使用 a 命令增加字符 what。使用 u 命令恢复回去。

- ⑰ 将光标移到 vim 上，使用 cw 命令将其修改成 VIM，按下 Esc 键，使用 . 重复命令完成其他 vim 的修改。
- ⑱ 使用 >> 命令查看该行效果，使用 << 命令查看该行效果。
- ⑲ 使用末行模式：%s/VIM/vim/g 修改所以 VIM 为 vim。
- ⑳ 使用末行模式：1,5 s/vim/VIM 命令，查看效果，使用 u 恢复回去。
- ㉑ 使用末行模式：1,5 s/vim/VIM/g 命令，查看效果，使用 u 恢复回去。
- ㉒ 使用末行模式：2,\$ s/vim/VIM/c 命令，查看修改效果。
- ㉓ 使用末行模式：s/^/XXX，在该行首添加 XXX，使用 u 恢复回去。
- ㉔ 使用末行模式：s/\$/YYY，在该行末添加 YYY，使用 u 恢复回去。
- ㉕ 使用末行模式：%s/languages.\$/MMM，使用 u 恢复回去。
- ㉖ 使用末行模式：%s/usr\share/USR\SHARE/g，查看修改效果，使用 u 恢复回去。
- ㉗ 光标移到任一行，使用 ~ 进行大小写切换。
- ㉘ 按下 Esc 键，按下：1, 3 w vim.tmp 将文件 1 行~3 行保存在 vim.tmp 文件中。使用 r vim.tmp 读入文件。
- ㉙ 使用末行模式：g/^\$/d，删除所有空行。
- ㉚ 使用 v 命令，使其进入可视化操作，移动光标三行，按下 “:”，这时行末会出现：'<,'> 表示光标范围，在其后添加 s/vim/VIM/g，进行替换操作。
- ㉛ 使用 s 修改单个字符。
- ㉜ 编写一个 C 语言文件，使用 ==、30==、=G 进行文件排版，使用 % 检查 { 和 (的匹配。

11.2 gcc

gcc 是 Linux 操作系统下 C 语言、C++ 语言的开源编译器。

11.2.1 gcc 简要说明

1. gcc 支持的文件格式

表 11-27 列出了 gcc 支持的文件格式及其说明。

表 11-27 gcc 支持的文件格式

文件后缀	文件说明
.c	C 源程序

续表

文件后缀	文件说明
.C、.cc、.cxx、.cpp	C++源程序
.m	Objective C 源程序
.i	预处理后的 C 文件
.ii	预处理后的 C++文件
.s	汇编语言源程序
.S	汇编语言源程序
.h	预处理文件
.o	目标文件 (Object file)
.a	归档库文件 (Archive file)

2. gcc 的组成

gcc 一般安装在/usr/bin 目录下。

gcc 是一组编译工具的总称，包含如下工具：

- ① C 编译器 cc、ccl、cclplus、gcc；
- ② C++编译器 c++、cclplus、g++；
- ③ 源码预处理程序 cpp、cpp0；
- ④ 库文件 libgcc.a、libgcc_eh.a、libgcc_s.so、libiberty.a、libstdc++、libsupc++.a。

3. gcc 编译过程

在 gcc 中，生成.o 和.obj 的过程叫做编译，把大量的.o 文件合成执行文件叫做链接。任何一个可执行程序从源代码到可执行的二进制程序之中都要经过固定的几步：

- ① 预编译 (Pre-Processing)：这一步完成对预编译代码的处理。
- ② 编译 (Compiling)：将源代码编译成汇编代码。
- ③ 汇编 (Assembling)：将汇编代码汇编成目标文件。
- ④ 链接 (Linking)：将目标代码和所需要的库链接成一个完整的可执行程序。

集成开发环境 (IDE) 自动协助开发完成这几步，如 VC++。在 Linux 下，如果使用命令行开发工具 (gcc、ld、ar) 等，需要用户手工调用这些命令来完成这几个步骤，而使用 Makefile 工具可以完成自动化编译的功能。

4. gcc 编译过程说明

gcc 在构建应用程序里，会调用不同的应用程序完成每一步。gcc 所做操作步骤如下：

- ① gcc 调用 cpp 进行预处理。
- ② gcc 调用 cc1 进行编译，会生成汇编代码。
- ③ gcc 调用 as 对汇编代码，生成扩展名为 .o 的目标文件。
- ④ gcc 调用 ld 来完成对所有目标文件的链接。

11.2.2 gcc 参数

1. gcc 常用选项列表

表 11-28 列出了 gcc 常用选项及其解释。

表 11-28 gcc 常用选项列表

选 项	解 释
-ansi	只支持 ANSI 标准的 C 语法。这一选项将禁止 GNU C 的某些特色，如 asm 或 typedef 关键词
-c	只编译并生成目标文件
-DMACRO	以字符串“1”定义 MACRO 宏
-DMACRO=DEFN	以字符串“DEFN”定义 MACRO 宏
-E	只运行 C 预编译器
-g	生成调试信息，GNU 调试器可利用该信息
-IDIRECTORY	指定额外的头文件搜索路径 DIRECTORY
-LDIRECTORY	指定额外的函数库搜索路径 DIRECTORY
-lLIBRARY	连接时搜索指定的函数库 LIBRARY
-o FILE	生成指定的输出文件。用来生成名称为 FILE 的可执行文件
-O0	不进行优化处理
-O 或 -O1	优化生成代码
-O2	进一步优化
-O3	比 -O2 更进一步优化，包括 inline 函数
-shared	此选项表明尽量使用动态链接库
-static	禁止使用动态链接库
-UMACRO	取消对 MACRO 宏的定义
-w	不生成任何警告信息
-Wall	生成所有警告信息

2. gcc 常用选项参数

gcc 选项参数重点应掌握-[Ii]、-[Ll]、-D，其他参数了解即可。

3. gcc 语法格式

gcc 的语法格式如下：

```
gcc [ option | filename ]...
g++ [ option | filename ]...
```

其中 option 为 gcc 使用时的选项，而 filename 为 gcc 要处理的文件名称。

4. gcc 选项之-[Ii]

-I 选项的作用为链接包含目录，此时源程序自动会在此目录下找相关头文件。在一个 gcc 命令中可以用多个-I，-I 表示包含目录，-i 表示包含特定的头文件。

此参数使用方法如下：

```
gcc foo.c -I /home/xiaowp/include -o foo
```

5. gcc 选项之-[Ll]

-L 选项的作用为链接包含相关链接库，此时源程序自动会在此目录下找相关库文件。-L 表示查找链接库目录，-l 表示查找链接库文件。

此参数使用方法如下：

```
gcc foo.c -L/home/hxy/lib -lfoo -o foo
```

6. gcc 选项之宏

-DMACRO 定义宏 MACRO，定义宏 MACRO 的值为字符串“1”，使用方法如下：

```
gcc test_m.c -D__DEBUG -o test_m
```

-DMACRO=defn 定义宏 MACRO 的内容为 defn。使用方法如下：

```
gcc test_m.c -D__DBG_NAME=hello -o test_m
```

-UMACRO 取消宏 MACRO。

-U 选项在所有的-D 选项之后处理，但是优先于任何 include 指令。

7. gcc 选项之代码优化

-O -O1：1 级优化。

-O2：2 级优化。除了涉及空间和速度交换的优化选项，执行几乎所有的优化工作。和-O 选项比较，这个选项既增加了编译时间，也提高了生成代码的运行效果。

-O3 优化得更多，除了打开-O2 所做的一切，它还打开了 finline-functions 选项。

-O0 不优化。

如果指定了多个-O 选项，不管带不带数字，最后一个选项才是生效的选项。

8. cc 选项之-x

-x language 选项明确指出后面输入文件的语言为 language，而不是从文件名后缀得到的默认选择。这个选项应用于后面所有的输入文件，直到遇着下一个-x 选项。language 的可选值有 c、objective-c、c-header、c++、assembler 等。

例如，“`gcc -x c++ hello.c`”表示强制用 `c++` 编译器来编译 `hello.c`。

关闭任何对语种的明确说明，因此依据文件名后缀处理后面的文件，就像是从未使用过 `-x` 选项一样。

关闭此选项方法如下：

```
-x none
```

9. gcc 选项之-c

`-c` 选项的作用为编译或汇编源文件，但是不做链接。

此时编译器输出对应于源文件的目标文件。默认情况下，`gcc` 通过用“`.o`”替换源文件名后缀“`.c`”、“`.i`”、“`.s`”等，产生目标文件名。可以使用 `-o` 选项选择其他名字。

`gcc -c` 选项忽略后面任何无法识别的输入文件（它们不需要编译或汇编）。

此选项使用方法如下：

```
gcc -c hello.c
```

10. gcc 选项之-S

`-S` 选项的作用为编译后即停止，不进行汇编编译，相当于编译源码，只生成汇编代码。

对于每个输入的非汇编语言文件，输出文件是汇编语言文件。默认情况下，`gcc` 通过用“`.o`”替换源文件名后缀“`.c`”、“`.i`”等，产生目标文件名。可以使用 `-o` 选项选择其他名字。

此选项使用方法如下：

```
gcc -S hello.c -o hello.s
```

11. gcc 选项之-E

`-E` 选项的作用为预处理后即停止，不进行编译。

预处理后的代码送往标准输出，`gcc` 忽略任何不需要预处理的输入文件。

此参数使用方法如下：

```
gcc -E hello.c -o hello.i
```

12. gcc 选项之-v

`-v` 选项显示执行编译阶段的命令，同时显示编译器驱动程序、预处理器、编译器的版本号。

此参数使用方法如下：

```
gcc -v
```

13. gcc 选项之-o

`-o file` 选项的作用为指定输出文件为 `file`。

该选项不在乎 gcc 产生什么输出，可以是可执行文件、目标文件、汇编文件、预处理后的 C 代码。

此选项使用方法如下：

```
gcc -E hello.c -o hello.i
gcc -c hello.i -o hello.o
gcc hello.c -o hello
```

14. gcc 选项之-pipe

-pipe 选项的作用为在编译过程的不同阶段间使用管道而非临时文件进行通信。

将源代码变成可执行文件的过程中，需要经过许多中间步骤，包含预处理、编译、汇编和链接，这些过程实际上是由不同的程序负责完成的。大多数情况下，gcc 可以为 Linux 程序员完成所有的后台工作，自动调用相应程序进行处理。

gcc 在处理每一个源文件时，最终都需要生成好几个临时文件才能完成相应的工作，从而无形中导致处理速度变慢。例如，gcc 在处理一个源文件时，可能需要一个临时文件来保存预处理的输出、一个临时文件来保存编译器的输出、一个临时文件来保存汇编器的输出，而读写这些临时文件显然需要耗费一定的时间。当软件项目变得非常庞大的时候，花费在这上面的代价可能会变得很沉重。

解决上述问题的办法是，使用 Linux 提供的一种更加高效的通信方式——管道。其中一个程序的输出将被直接作为另一个程序的输入，这样就可以避免使用临时文件，但编译时却需要消耗更多的内存。

此参数使用方法如下：

```
gcc -pipe foo.c -o foo
```

15. gcc 选项之警告说明

gcc 包含完整的出错检查和警告提示功能，它们可以帮助 Linux 程序员写出更加专业和优美的代码。

-pedantic 选项编译警告代码。

打开则完全按照 ANSI C 标准，提供全部的警告诊断，有警告后停止编译。

-ansi 选项支持符合 ANSI 标准的 C 程序。

这样会关闭 GNU C 中某些不兼容 ANSI C 的特性，如 asm、inline 和 typedef 关键字。同时，开启不受欢迎和极少使用的 ANSI trigraph 特性，以及禁止 “\$” 成为标识符的一部分。该选项与 pedantic 的区别在于，只是警告，如果需要停止编译，仍然需要打开 -pedantic 选项。

16. gcc 选项之警告

-Wall 选项打开所有编译警告。

此参数使用方法如下：

```
gcc -Wall illcode.c -o illcode
```

`-Werror` 选项视警告为错误，出现任何警告即放弃编译。

此参数使用方法如下：

```
gcc -Wall -Werror illcode.c -o illcode
```

`-w` 选项禁止输出警告信息。

17. gcc 选项之-g

`-g` 选项以操作系统的本地格式（`stabs`、`COFF`、`XCOFF` 或 `DWARF`）产生调试信息。

`gdb` 能够使用这些调试信息，此参数是进行 `gdb` 调试的必备条件。为了可以产生调试信息，编译时需要带 `-g` 选项。

此参数使用方法如下：

```
gcc hello.c -g -o hello
```

`-pg` 选项产生额外代码，用于输出 `profile` 信息，供分析程序 `gprof` 使用。

所有调试选项会使最终输出文件大小急剧增加，在最后发布时，可以使用 `strip` 命令把调试信息去掉。

`strip` 使用方法如下：

```
strip hello
```

11.3 Makefile

一个软件项目通常包含多个源码文件，每个源代码的编译和可执行文件的链接都要书写大量的命令，如 Linux 下要大量调用 `gcc` 来处理。

如果用 IDE 开发环境，编译和链接一般都由 IDE 自动完成，但绝大部分 Linux 和开源项目并不使用 IDE，而是使用 `gcc` 之类命令行工具来编译。

在一个项目中，代码通常都有引用关系。因此需要指定谁先编译，谁后编译，甚至是更复杂的功能操作。

`Makefile` 就是为解决上述问题而创造的，可以把 `Makefile` 理解成是一种由 `make` 程序进行解释的一种特殊脚本。

Linux 下几乎所有项目都是通过 `Makefile` 方式编译的，如 MySQL、Apache 和操作系统本身，因此在 Linux 下进行开发需要掌握 `Makefile` 的编写和使用。

11.3.1 Makefile 简介

Makefile 是定义整个工程的编译规则，完成工程自动化编译的工具。

1. Makefile 与 Shell 脚本的异同

Makefile 与 Shell 脚本的相同点有：两者都是文本文件格式的脚本，都可以执行 Shell 命令，都可以定义变量和条件控制语句（使用格式上有差别）。

Makefile 与 Shell 脚本的不同点有：解释器不同，Shell 脚本是由对应 Shell 程序解释，而 Makefile 是由 make 程序解释；格式不一样，Shell 脚本以命令行为基本单位，Makefile 以规则为基本单位；Shell 脚本只要有执行权限即可直接执行，Makefile 必须要用 make 来显式调用才行，本身不需要执行权限。

2. Makefile 相对 Shell 脚本的优点

在开发领域，Makefile 的优势在于 Makefile 具有自动推导、判断源码依赖关系的功能。Makefile 可以使用隐含规则来简化 Makefile 的编写，但这样也会带来 Makefile 可阅读性的下降。

3. Makefile 规则

Makefile 是以规则为单位的。Makefile 的规则由规则的目标、规则的依赖和规则的命令行三部分组成。

(1) Makefile 规则举例

下面是 Makefile 规则的定义格式。

```
TARGET : PREREQUISITES
        COMMAND
```

对上文 Makefile 定义格式的说明如下。

TARGET：规则的目标。可以是文件名，也可以是一个 make 的执行动作。

PREREQUISITES：规则的依赖，由一个文件和多个文件组成，也可以没有任何文件。

COMMAND：规则的命令行。由一个或多个命令组成，每一个命令占一行，必须由[TAB]字符开始。

(2) Makefile 规则说明

在 Makefile 中，编译文件时规则通常包含三个部分，包括依赖文件（规则的依赖）、规则方法（规则命令行）和目标文件（规则的目标）。

举例说明如下：

```
main.o : main.c defs.h
```



```
cc -c main.c
```

上例中目标文件为 main.o，依赖文件为 main.c 和 defs.h，规则方法为 cc -c main.c。

(3) Makefile 构成五部分

Makefile 由下面五部分组成：

- ① 显式规则：直接推导的规则，只需一次推导过程。
- ② 隐晦规则：间接推导的规则，需要一次以上的推导过程。
- ③ 变量定义。
- ④ 文件指示。
- ⑤ 注释。

(4) Makefile 文件样例

下面以一个 Makefile 文件来说明 Makefile 组成的五部分。

```
#This is makfile file      ---注释
include Custom.defines     ----文件指示
cc=gcc                    ----cc 为变量
foo.o:                    ----显示规则
    cc foo.c -o foo.o
foo : foo.o               ---隐含规则
cc -o foo foo.o $(CFLAGS) (LDFLAGS)
```

(5) Makefile 组成的五部分详细说明

下面是 Makefile 组成五部分的详细解释。

① 显式规则：它描述了在何种情况下，如何更新一个或者多个被称为目标的文件（Makefile 的目标文件）。书写 Makefile 时，需要明确地给出目标文件、目标的依赖文件列表以及更新目标文件所需要的命令。有些规则没有命令，这样的规则只是纯粹地描述了文件之间的依赖关系。

② 隐含规则：它是 make 根据一类目标文件（典型的是根据文件名的后缀）而自动推导出来的规则。make 根据目标文件的名，自动产生目标的依赖文件并使用默认的命令来对目标进行更新（建立一个新规则）。

③ 变量定义：使用一个字符或字符串代表一段文本串，当定义一个变量以后，Makefile 后续在需要使用此文本串的地方，通过引用这个变量来实现对文本串的使用。

④ Makefile 指示符：指示符指明 make 程序读取 Makefile 文件过程中所要执行的一个动作。其中包括：读取一个文件，读取给定文件名的文件，将其内容作为 Makefile 文件的一部分；决定（通常是根据一个变量的值）处理或者忽略 Makefile 中的某一特定部分。

⑤ 注释：Makefile 中“#”字符后的内容被作为注释内容（和 Shell 脚本一样）处理。如

果此行的第一个非空字符为“#”，那么此行为注释行。注释行的结尾如果存在反斜线“\”，那么下一行也被作为注释行。一般在书写 Makefile 时，推荐将注释作为一个独立的行，而不要和 Makefile 的有效行放在一行中书写。当在 Makefile 中需要使用字符“#”时，可以使用反斜线“\”加“#”（如\#）来实现，其表示将“#”作为一个字符而不是注释的开始标志。

4. Makefile 与 gcc 的分工

Makefile 负责框架，组织编译过程，而 gcc 只负责编译。

Makefile 同样支持其他编译器。

5. Makefile 的通配符

Makefile 中表示文件名时可使用通配符。可使用的通配符有：“*”、“?”和 “[...]”，含义与正则表达式相同。

“\”在 Makefile 文件中为续行或转义。

11.3.2 Makefile 语法

1. Makefile 的返回值

make 命令执行后有三个退出码：

- ① 0：表示成功执行。
- ② 1：如果 make 运行时出现任何错误，返回 1。
- ③ 2：如果使用了-q 选项，并且一些目标文件不需要更新，那么返回 2。

2. Makefile 命令行@作用

Makefile 会自动显示命令行，@表示不显示此命令行。

“@echo 正在编译 XXX 模块.....”会输出“正在编译 XXX 模块.....”。

“echo 正在编译 XXX 模块.....”会输出“echo 正在编译 XXX 模块.....”、“正在编译 XXX 模块”两行。

3. Makefile 命令行开始的 Tab 键

一个规则可以有多个命令行，每一条命令占一行。每一个命令行必须以 [Tab] 字符开始，[Tab] 字符告诉 make 此行是一个命令行，make 对这样的行按照命令行进行解释，完成相应的动作。这也是书写 Makefile 中容易产生的，而且比较隐蔽的错误。

4. Makefile 命令行头-作用

Makefile 的命令行前加一个减号“-”（在 Tab 键之后），标记为不管命令出不出错，都认为是成功的。

举例说明如下：

```
clean:
    -rm -f *.o
```

无论 make clean 是否执行成功，都认为执行成功。

5. Makefile 变量组成

Makefile 变量包括预定义变量，自定义变量和用户环境变量三种。其中用户环境变量也可广义地归为 Makefile 预定义变量这一类。

6. Makefile 符号预定义变量

Makefile 有许多预定义的变量，这些变量具有特殊的含义，可在规则中使用。

除这些变量外，GNU make 还将所有的环境变量作为自己的预定义变量。

表 11-29 列出了在 Makefile 中的符号预定义变量及其含义说明。其中，在实际工作中使用最频繁的符号预定义变量有\$@和\$<两种。假设存在规则 main:main.o func.o，则\$@表示main，\$<表示main.o，而\$+、\$?、\$^表示main.o func.o。

表 11-29 Makefile 的符号预定义变量

变 量	含义说明
\$+	所有的依赖文件，以空格分开，并以出现的先后为序，可能包含重复的依赖文件
\$?	所有依赖文件，以空格分开，这些依赖文件的修改日期比目标的创建日期晚
\$^	所有的依赖文件，以空格分开，不包含重复的依赖文件
\$@	目标的完整名称
\$<	第一个依赖文件的名称
\$%	仅当目标是归档成员（函数库文件）时，该变量表示目标的归档成员名称。例如，如果目标名称为mytarget.so(image.o)，则\$@为mytarget.so，而\$%为image.o。如果目标不是归档成员，此值为空

7. 与编译相关的预定义变量

表 11-30 列出了在 Makefile 中与编译相关的预定义变量及其含义说明。

表 11-30 与编译相关的预定义变量

与编译相关的预定义变量	含义说明
AR	归档维护程序的名称，默认值为 ar
ARFLAGS	归档维护程序的选项
AS	汇编程序的名称，默认值为 as
ASFLAGS	汇编程序的选项
CC	C 编译器的名称，默认值为 cc
CCFLAGS	C 编译器的选项
CPP	C 预编译器的名称，默认值为 \$(CC) -E
CPPFLAGS	C 预编译的选项
CXX	C++ 编译器的名称，默认值为 g++
CXXFLAGS	C++ 编译器的选项

8. Makefile 自定义变量

(1) 变量名定义规则

变量名定义规则为：变量名是不包括“:”、“#”、“=”、前置空白和尾空白的任何字符串；变量名是大小写敏感的；变量可递归引用。

(2) 变量的四种定义方式

① 使用=或 define 来定义，这种方式下，前面变量使用后面的变量时，Makefile 推导时会自动先对后面变量进行赋值之后，再对前面的变量进行赋值。

② 使用:=号定义变量，这种方式下前面，变量使用后面的变量时，后面的变量在当前使用时为空值。

③ 使用+=对变量追加内容。

④ 使用?=对变量进行定义，如果前面变量未定义则定义，否则什么也不做。

(3) 变量定义示例

用 vi 编辑器编写一个 Makefile 文件，此文件头为常见的 Makefile 使用模板，由 first 变量开始为变量定义内容。

Makefile 内容如下：

```
#用户头文件
INCLDIR=-I$(HOME)/include
#用户目录
LIBDIR=-L$(HOME)/lib
#定义后缀规则
.SUFFIXES: .sqc .c.o .o
#自动推导编译.c 源文件
.c.o:
    gcc $(INCLDIR) $(LIBDIR) -c $<
first=$(second)
second=hello
sunday:=$(monday)moon
monday:=monday
sky:=red
sea?=RED
sky?=blue
#下面两种定义方式等价
xobj=x.o xx.o
xobj+=xxx.o
yobj=y.o yy.o
yobj:=$(yobj) yyy.o
all:firstaim sundayaim skyaim xyaim
firstaim:
    @echo "test ="
    @echo $(first)
sundayaim:
```

```
@echo "test :="
@echo $(sunday)
skyaim:
@echo "test ?="
@echo $(sky)
@echo $(sea)
xyaim:
@echo "test +=="
@echo $(xobj)
@echo $(yobj)
clean:
-rm -f *.o
```

在命令行执行 `make all`，执行结果如下：

```
test =
hello
test :=
moon
test ?=
red
RED
test +=
x.o xx.o xxx.o
y.o yy.o yyy.o
```

根据执行结果说明如下：

- ① 使用 “=” 对变量进行赋值时，`first` 变量使用后面的 `second` 变量，`second` 变量先进行赋值，然后再对 `first` 进行赋值。
- ② 使用 “:=” 对变量赋值时，`sunday` 变量使用后面的 `monday` 变量，`monday` 变量对 `sunday` 变量不起作用（其值为空）。
- ③ 使用 “?=” 对变量进行赋值时，`sky` 变量进行了两次定义，但只有第一次的定义有效。
- ④ 使用 “+=” 对变量进行赋值时，`xobj` 变量对原有内容保留，对新的赋值进行了追加。

9. Makefile 的用户环境变量

即在用户环境中 `.profile` 中定义并生效的用户环境变量，这些用户环境变量，`Makefile` 当做预定义变量处理。

10. Makefile 后缀规则两种推导方式

针对编译某一类的文件，`Makefile` 可以定义隐含（通配）规则来完成推导。`Makefile` 的隐含规则可以通过两种方式去实现，一种为后缀规则，另一种为模式规则。

（1）Makefile 的后缀规则

`Makefile` 的后缀规则，必须首先由 `.SUFFIXES` 来进行声明支持后缀的类型，如 `.SUFFIXES: .sqc .c`。

后缀规则（Suffix Rule）是定义隐含规则的老风格方法。后缀规则定义将一个具有某个后缀的文件（如.c 文件）转换为具有另外一种后缀的文件（如.o 文件）的方法。每个后缀规则以两个成对出现的后缀名定义，如将“.c”文件转换为“.o”文件的后缀规则可定义如下：

```
.c.o:
    $(CC) $(CFLAGS) $(CPPFLAGS) -c -o $@ $<
```

(2) Makefile 的模式规则

在 Makefile 中，模式规则（pattern rules）更加通用，因为可以利用它定义更加复杂的依赖性规则。模式规则看起来非常类似于正则规则，但在目标名称的前面多了一个%号，同时可以用来定义目标和依赖文件之间的关系，例如，下面的模式规则定义了如何将任意一个“x.c”文件转换为“x.o”文件：

```
%.o:%.c
    $(CC) $(CFLAGS) $(CPPFLAGS) -c -o $@ $<
```

11.3.3 Makefile 的运行

1. Makefile 的文件名

Makefile 的一般默认文件名为 makefile 或 Makefile，对上述两种文件进行编译时，只需执行“make 规则目标名”即可完成编译。

非上述两种文件名，如 makefile.aix，要完成编译，就需执行“make -f makefile.aix 规则目标名”，才能完成编译。

2. Makefile 命令行参数

表 11-31 列出了 Makefile 在命令行执行的参数及其含义说明。命令行下，使用最多的是 -f 参数。

表 11-31 Makefile 命令行参数

命令行选项	含 义
-C DIR	在读取 makefile 之前改变到指定的目录 DIR
-f FILE	以指定的 FILE 文件作为 makefile
-h	显示所有的 make 选项
-i	忽略所有的命令执行错误
-I DIR	当包含其他 makefile 文件时，可利用该选项指定搜索目录
-n	只打印要执行的命令，但不执行这些命令
-p	显示 make 变量数据库和隐含规则
-s	在执行命令时不显示命令
-w	在处理 makefile 之前和之后，显示工作目录
-W FILE	假定文件 FILE 已经被修改

3. Makefile 的编译规则

- ① 如果这个工程没有编译过，那么所有 c 文件都要被编译并被链接。

② 如果这个工程的某几个 c 文件被修改，那么只编译被修改的 c 文件，并链接目标程序。

③ 如果这个工程的头文件被改变了，那么需要编译引用了这几个头文件的 c 文件，并链接目标程序。

4. Makefile 的工作流程

Makefile 的工作流程如下：

- ① 读入所有的 Makefile。
- ② 读入被 include 的其他 Makefile。
- ③ 初始化文件中的变量。
- ④ 推导隐晦规则，并分析所有规则。
- ⑤ 为所有的目标文件创建依赖关系链。
- ⑥ 根据依赖关系，决定哪些目标要重新生成。
- ⑦ 执行生成命令。

11.3.4 Makefile 的扩展话题

1. Makefile 中的条件执行

很少使用，本节只列出，使用时请自行查阅文档。

```
ifdef VARIABLE
ifndef VARIABLE
ifeq (A,B)
ifeq "A" "B"
ifeq 'A' 'B'
ifneq (A,B)
ifneq "A" "B"
ifneq 'A' 'B'
else
endif
```

2. Makefile 中的函数

很少使用，本节只列出，使用时请自行查阅文档。

文本处理函数：subst、patsubst、strip、indstring、filter、filter-out、sort、word、words、wordlist、firstword。

文件名处理函数：dir、notdir、suffix、basename、addsuffix、addprefix、join、wildcard。

其他函数：error、warning、shell、origin、foreach、call、if、eval、vaule。

3. Makefile 的约定

所有的 Makefile 中最好包含这样一行：SHELL= /bin/sh,其目的是为了避免变量“SHELL”在有些系统上可能继承同名的系统环境变量而导致的错误。

小心处理后缀规则和隐含规则。不同 make 可识别后缀和隐含规则可能不同，它可能会导致混乱或者错误。因此，在特定 Makefile 中明确限定可识别的后缀是一个不错的主意，在 Makefile 中应该这样做：

```
.SUFFIXES:
.SUFFIXES: .c .o
```

第一行首先取消掉 make 默认的可识别后缀列表，第二行重新指定可识别的后缀列表。

11.4 gdb

GNU 的调试器称为 gdb，该程序是一个交互式工具，工作在字符模式。在 X Window 系统中，有一个 gdb 的前端图形工具，称为 xxgdb。gdb 是功能强大的调试程序，可完成如下的调试任务：设置断点、监视程序变量的值、程序的单步执行、修改变量的值。

在使用 gdb 调试程序之前，必须使用 -g 选项编译源文件。可在 Makefile 中对 CFLAGS 变量增加 -g 选项，增加方法为 CFLAGS+=-g。

运行 gdb 调试程序时，通常使用如下的命令：gdb progname。

11.4.1 gdb 语法

1. gdb 命令分类

在 gdb 提示符处键入 help，将列出命令的分类，主要的分类有表 11-32 列出的九种。键入 help 后跟命令的分类名（如 help stack），可获得该类命令的详细清单。

表 11-32 gdb 命令分类表

命令分类名称	说 明
aliases	命令别名
breakpoints	断点定义
data	数据查看
files	指定并查看文件
internals	维护命令
running	程序执行
stack	调用栈查看
status	状态查看
tracepoints	跟踪程序执行

2. gdb 主要命令列表

表 11-33 列出了 gdb 的主要命令及其解释说明。

表 11-33 gdb 主要命令列表

gdb 命令		解释说明
backtrace		显示程序中的当前位置和表示如何到达当前位置的栈跟踪，可以简写为 bt
breakpoint		在程序中设置一个断点
cd		改变当前工作目录
clear		删除刚才停止处的断点
commands		命中断点时，列出将要执行的命令
continue		从断点开始继续执行
delete		删除一个断点或监测点。也可与其他命令一起使用
display		显示变量和表达式
down		下移栈帧，使得另一个函数成为当前函数
frame		选择下一条 continue 命令的帧
info		显示与该程序有关的各种信息
jump		在源程序中的另一点开始运行
kill		异常终止在 gdb
list		列出相应于正在执行的程序的源文件内容
next		执行下一个源程序行，从而执行其整体中的一个函数
print		显示变量或表达式的值
pwd		显示当前工作目录
quit		退出 gdb
reverse-search		在源文件中反向搜索正则表达式
run		执行该程序
search		在源文件中搜索正则表达式
set variable		给变量赋值
signal		将一个信号发送到正在运行的进程
step		执行下一个源程序行，必要时进入下一个函数
undisplay		display 命令的反命令，不要显示表达式
until		结束当前循环
up		上移栈帧，使另一函数成为当前函数
watch		在程序中设置一个监测点（即数据断点）
whatis		显示变量或函数类型
where		打印所有堆栈帧的栈信息

3. gdb 常用命令使用方法

表 11-34 列出了 gdb 常用命令使用方法及其解释说明。

表 11-34 gdb 常用命令使用方法

gdb 常用命令		解释说明
break NUM		在指定的行上设置断点
clear		删除设置在特定源文件、特定行上的断点。其用法为 clear FILE:NUM
continue		继续执行正在调试的程序
display EXPR		每次程序停止后，显示表达式的值

续表

gdb 常用命令	解释说明
file FILE	装载指定的可执行文件进行调试
help NAME	显示指定命令的帮助信息
info break	显示当前断点清单，包括到达断点处的次数等
info files	显示被调试文件的详细信息
info func	显示所有的函数名称
info local	显示当前函数中的局部变量信息
info prog	显示被调试程序的执行状态
info var	显示所有的全局和静态变量名称
kill	终止正被调试的程序
list	显示源代码段
make	在不退出提示符的情况下编译源代码
next	在不单步执行进入其他函数的情况下，向前执行一行源代码
print EXPR	显示表达式 EXPR

4. gdb 命令详解

（1）列出文件清单 list

```
(gdb) list line1,line2
```

（2）执行程序

要想运行准备调试的程序，可使用 run 命令，在它后面可以跟随发给该程序的任何参数。

如果使用不带参数的 run 命令，gdb 就再次使用前一条 run 命令的参数。

利用 set args 命令就可以修改发送给程序的参数，而使用 show args 命令就可以查看其默认参数的列表。

```
(gdb) set args -b -x
(gdb) show args
```

backtrace 命令为堆栈提供向后跟踪功能，backtrace 命令产生一张列表，包含着从最近的过程开始的所有有效过程和调用这些过程的参数。

（3）显示数据

利用 print 命令可以打印出各个变量的值，还可以显示被调试的语言中任何有效的表达式。print 使用方法说明如下。

① 显示程序中的变量：

```
(gdb) print p (p 为变量名)
```

② 显示程序中函数调用：

```
(gdb) print find_entry(1,0)
```

③ 显示程序中数据结构和复杂对象：

```
(gdb) print *table_start
$8={e=reference=' \000',location=0x0,next=0x0}
```

④ 利用 `whatis` 命令可以显示某个变量的类型：

```
(gdb) whatis p
type = int *
```

(4) 断点 (breakpoint)

`break` 命令（可以简写为 `b`）可以用来在调试的程序中设置断点。

该命令有如下四种形式：

① `break line-number`，使程序恰好在执行给定行之前停止。

② `break function-name`，使程序恰好在进入指定的函数之前停止。

③ `break line-or-function if condition`，如果 `condition`（条件）是真，程序到达指定行或函数时停止。

④ `break routine-name`：在指定例程的入口处设置断点。

如果该程序是由很多源文件构成的，可以在各个源文件中设置断点，而不是在当前的源文件中设置断点，其方法如下：

```
(gdb) break filename:line-number
(gdb) break filename:function-name
```

要想设置一个条件断点，可以利用 `break if` 命令，使用方法如下所示：

```
(gdb) break line-or-function if expr
(gdb) break 46 if testsize==100
```

从断点继续运行可使用 `countinue`（简写为 `c`）命令。

(5) 断点的管理

① 显示当前 `gdb` 的断点信息方法如下：

```
(gdb) info break
```

会以如下的形式显示所有的断点信息：

```
Num Type Disp Enb Address What
1 breakpoint keep y 0x000028bc in init_random at qsort2.c:155
2 breakpoint keep y 0x0000291c in init_organ at qsort2.c:168
(gdb)
```

② 删除指定的某个断点方法如下：

```
(gdb) delete breakpoint 1
```

该命令将会删除编号为 1 的断点，如果不带编号参数，将删除所有断点。

```
(gdb) delete breakpoint
```

③ 禁止使用某个断点方法如下：

```
(gdb) disable breakpoint 1
```

该命令将禁止断点 1，同时断点信息的 (Enb) 域将变为 n。

④ 允许使用某个断点方法如下：

```
(gdb) enable breakpoint 1
```

该命令将允许断点 1，同时断点信息的 (Enb) 域将变为 y。

⑤ 清除源文件中某一代码行上的所有断点方法如下 (number 为源文件的行号)：

```
(gdb) clean number
```

(6) 变量的检查和赋值

变量的检查和赋值相关命令如下：

① whatis：识别数组或变量的类型。

② ptype：比 whatis 的功能更强，它可以提供一个结构的定义。

③ set variable：将值赋予给变量。

④ print：除了显示一个变量的值外，还可以用来赋值。

(7) 单步执行

单步执行的相关命令如下：

① next：不进入函数的单步执行。

② step：进入函数的单步执行。

③ 如果已经进入了某函数，当想退出该函数返回到它的调用函数中时，可使用命令 finish。

(8) 函数的调用

call 用来调用和执行一个函数，使用方法如下：

```
(gdb) call gen_and_sork( 1234,1,0 )
```

```
(gdb) call printf( "abcd" )
```

```
$1=4
```

finish 结束执行当前函数，显示其返回值 (如果有的话)。

(9) 机器语言工具

有一组专用的 gdb 变量可以用来检查和修改计算机的通用寄存器，gdb 提供了目前每一台计算机中实际使用的 4 个寄存器的标准名字，其名字如下：

- ① \$pc : 程序计数器。
- ② \$fp : 帧指针 (当前堆栈帧)。
- ③ \$sp : 栈指针。
- ④ \$ps : 处理器状态。

(10) 源文件的搜索

search text: 该命令可显示在当前文件中包含 text 字符串的下一行。

reverse-search text: 该命令可以显示包含 text 的前一行。

(11) 命令的历史

为了允许使用历史命令, 可使用 set history expansion on 命令。

```
(gdb) set history expansion on
```

11.4.2 gdb 调试

1. gdb 调试范例

下面是一个有错误的 C 语言源程序 test.c。

```
#include <stdio.h>
#include <stdlib.h>
static char buff [256];
static char* string;
int main ()
{
    printf ("Please input a string: ");
    gets (string);
    printf ("\nYour string is: %s\n", string);
    return 0 ;
}
```

上面这个程序非常简单, 其目的是接受用户的输入, 然后将用户的输入打印出来。该程序使用了一个未经过初始化的字符串 string, 因此, 编译并运行之后, 将出现 “Segment Fault” 错误。调试方法如下:

```
$ gcc -o test -g test.c
$ ./test
Please input a string: asfd
Segmentation fault (core dumped)

$gdb ./test    (效果同$(gdb):file ./test)
$run    (执行程序)
$where(或bt)    (查看 core 掉程序的行数和代码信息)
$print string    (打印变量信息)
$break 8    (设置断点)
$info break    (查看断点信息)
$run
```

```
$set variable string=buff (给指针变量赋值)
$n(next) (单条语句执行)
$c(continue)
```

对上述调试过程解释如下：

- ① 运行 `gdb./test` 命令，装入 `test` 可执行文件。
- ② 使用 `where`（或 `bt`）命令查看程序出错的地方。
- ③ 利用 `list` 命令查看调用 `gets` 函数附近的代码。
- ④ 唯一能够导致 `gets` 函数出错的原因就是变量 `string`，用 `print` 命令查看 `string` 的值。
- ⑤ 在第 8 行处设置断点，程序重新运行到第 8 行处停止，这时可以用 `set variable` 命令修改 `string` 的取值。
- ⑥ 然后继续运行，将看到正确的程序运行结果。

2. core 文件调试步骤

core 文件调试步骤如下：

- ① 用 `file core` 查看 `core` 的执行码。
- ② 用 `what core` 显示 `core` 的信息。
- ③ 用 `gdb --core =core.9128`（`core` 文件名）开始进行调试。

3. core 文件调试举例一

```
$gdb --core=core.9128 (指定加载 core 文件)
$bt (或 where) (查看 core 的行数及信息)
$file ./test (加载 core 的执行码，现约定 core 执行码为当前目录的 test)
$run (运行程序)
$bt(或 where) (查看 core 的行数及信息)
$list(l) (查看出错程序源码)
```

4. core 文件调试举例二

```
$gdb ./test (执行码) core.9128 (core 文件) (加载了执行码和 core 文件)
$bt(或 where) (查看 core 的行数及信息)
$list(l) (查看出错程序源码)
$run (运行程序)
$print var (打印变量信息，var 约定为此执行码的变量)
$break 8 (设置断点)
$info break (查看断点信息)
$n(next) (单条语句执行)
$c(continue)
```

5. 限制 core 文件的大小

在用户 `.profile` 或 `.bash_profile` 增加 `ulimit -c 30K`（30K 可以改为其他数字）语句来限制 `core` 文件的大小。

第 3 篇

Linux 进程

- ❖ 第 12 章 Linux 进程编程
- ❖ 第 13 章 Linux 线程编程
- ❖ 第 14 章 Linux 进程间通信——管道与信号
- ❖ 第 15 章 System V 进程间通信

学海聆听：

- 三思而后行。
- 善学者，假人之长以补其短。
- 培养一种良好的习惯受益终生。
- 许多事物的外延无法改变，然而内涵修炼能使事物大大增值。
- 人无远虑，必有近忧。
- 目标、态度、方法、意志、爱好。
- 知之者不如乐之者，乐知者不如好之者；兴趣是最好的老师。
- 理想是力量的源泉、智慧的摇篮、冲锋的战旗、斩棘的利剑。
- 知识改变命运，态度影响未来。
- 读书需用心，一字值千金。
- 懵懂而死，与草木同朽；悟道而生，是为永生。
- 行百里者半九十；从量变到质变，需要锲而不舍的努力。

第 12 章

Linux 进程编程

进程是程序的一次执行过程。在操作系统中，进程分为运行态、就绪态和等待态。正在使用 CPU 的进程为运行态；进程其他条件和资源就绪，只缺 CPU 资源的进程处于就绪态；等待态是进程除 CPU 外，还等待某些条件和资源的状态。在 Linux 操作系统中，采用时间片轮转法和抢占调度方式为每个进程分配 CPU 资源。进程是 Linux 环境编程的重要概念。

12.1 Linux 进程编程基本概念

12.1.1 登录

1. 用户登录名

登录 Linux 系统时，需要先键入用户登录名，然后键入用户密码，系统通过 `/etc/passwd`（口令文件）文件校验用户登录名和用户密码。口令文件中的登录项由 7 个以冒号分隔的字段组成，分别为登录名、加密口令、数字用户 ID（224）、数字组 ID（20）、注释字段、起始目录（`/home/stevens`），以及 Shell 程序（`/bin/sh`）。

2. 登录 Shell

登录后，系统先显示一些典型的系统信息，然后就可以向 Shell 程序键入命令。Shell 是一个命令行解释器，它读取用户输入，然后执行命令，用户通常使用终端，有时则通过 Shell 脚本文件向 Shell 进行输入。常用的 Shell 有：

- Bourne Shell (`/bin/sh`)
- C Shell (`/bin/csh`)
- Korn Shell (`/bin/ksh`)

系统从口令文件中登录项的最后一个字段确定应该执行哪一种 Shell。

Bourne Shell 是其最早版本，也是最流行的，C Shell 和 Korn Shell 是其后继开发和

升级的。三种 Shell 语法类似，功能基本相同。

12.1.2 文件和目录

1. 文件系统

Linux 文件系统是一种树形层次结构，包括目录和文件，目录的起点称为根（root），其名字是一个字符/。

目录（directory）是一个包含目录项的文件，在逻辑上，可以认为每个目录项都包含一个文件名，同时还包含说明该文件属性的信息。文件属性有文件类型、文件长度、文件所有者、文件的许可权、文件最后的修改时间等，可用 `stat` 和 `fstat` 函数返回一个包含所有文件属性的信息结构。

2. 文件名

目录中的各个名字称为文件名，斜线（/）不能出现在文件名中，斜线分隔是构成路径名。当创建一个新目录时，系统自动创建了两个文件名，分别为.（称为点）和..（称为点-点），点引用当前目录，点-点则引用父目录，在最高层次的根目录中，点-点与点相同。

3. 起始目录

登录时，工作目录设置为起始目录（home directory），该起始目录从口令文件中的登录项中取得。起始目录是创建用户时创建的，如 `test` 的起始目录为 `/home/test`。起始目录又称为主目录，用“~”可表示该用户的主目录。

4. 路径名

0 个或多个以斜线分隔的文件名序列构成路径名（pathname），以斜线开头的路径名称为绝对路径名（absolute pathname），否则称为相对路径名（relative pathname）。如当前目录为 `/home/test/hello`，`cd /home/test/hello/why` 是使用绝对路径方式到达该目录，`cd ../why` 是使用相对路径到达该目录，`cd ~/hello/why` 是通过主目录到达该目录。

5. 工作目录

每个进程都有一个工作目录（working directory），有时称为当前工作目录（current working directory）。所有相对路径名都从工作目录开始解释，进程可以用 `chdir` 函数更改其工作目录。

登录一个用户时就有一个工作目录，用 `cd` 命令可以改变工作目录，如登录 `test` 用户后敲入 `cd hello`，其工作目录变为 `/home/test/hello`。

12.1.3 输入和输出

1. 文件描述符

文件描述符是一个非负整数，内核用来标识一个特定进程正在访问的文件。当内核打开一个现存文件或创建一个新文件时，它就返回一个文件描述符。

每个进程在 Linux 内核中都有一个 `task_struct` 结构体来维护与进程相关的信息，称为进程描述符 (Process Descriptor)，又称为进程控制块 (PCB, Process Control Block)。 `task_struct` 中有一个指针指向 `files_struct` 结构体，称为文件描述符表，其中每个表项包含一个指向已打开文件的指针，如图 12-1 所示。

进程控制块 (`task_struct`) 指向文件指针的顺序为 `task_struct-> files(file_struct) ->fd 数组`， `fd` 数组大小决定了进程打开的最大文件个数。

用户程序不能直接访问内核中的文件描述符表，而只能使用文件描述符表的索引 (即 0、1、2、3 这些数字)，这些索引就称为文件描述符 (File Descriptor)，用 `int` 型变量保存。当调用 `open` 打开一个文件或创建一个新文件时，内核分配一个文件描述符并返回给用户程序，该文件描述符表项中的指针指向新打开的文件。当读写文件时，用户程序把文件描述符传给 `read` 或 `write`，内核根据文件描述符找到相应的表项，再通过表项中的指针找到相应的文件。

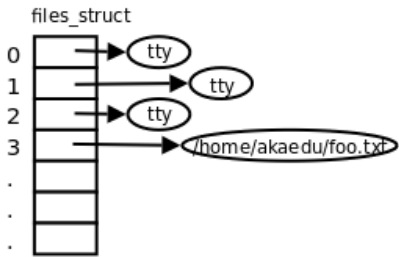


图 12-1 文件描述符表

2. 标准输入、标准输出和标准出错

每当运行一个新程序 (进程，包括 Shell 程序产生的进程) 时，进程自动打开三个文件描述符，即标准输入、标准输出以及标准出错。Shell 中 0 表示标准输入 (`stdin`)，1 表示标准输出 (`stdout`)，2 表示标准错误 (`stderr`)。

标准输入对应键盘，标准输出对应屏幕，标准错误同样对应屏幕。

Shell 都提供一种方法，使任何一个或所有这三个描述符都能重新定向到某一个文件，如 `ls>file.list`，就是将标准输出重新定向到名为 `file.list` 的文件上。

3. 不用缓存的 I/O

函数 `open`、`read`、`write`、`lseek` 以及 `close` 提供了不用缓存的 I/O，这些函数都是用文件描述符进行工作的。不用缓存的 I/O 函数一次读写操作完成一次系统调用，如当初级文件 I/O 写函数 `write` 所带的写大小参数太小时，会引起系统调用次数过多而造成系统效率低下。

4. 标准 I/O

标准 I/O 函数读写时无需关心缓存大小，操作系统对标准 I/O 函数自动分配缓存、使用缓存和管理缓存。使用标准 I/O 可无需担心如何选取最佳的缓存长度，如 `fread`、`fwrit`、`fprintf`、`fscanf`、`fgets` 等都是标准 I/O，标准 I/O 是不用缓存的 I/O 的发展，这些函数默认都使用了缓存。

12.1.4 程序与进程

1. 程序

可执行程序是存放在磁盘文件中的可执行文件，可使用 6 个 `exec` 函数中的一个由内核将程
深入浅出 Linux 工具与编程

序读入内存，并使其执行。

2. 进程和进程 ID

程序的执行实例被称为进程 (process)。进程空间 (包括代码段空间、数据区空间、运行的堆栈空间) 存在于内存中。CPU 执行的是进程代码段的代码，进程执行时间就是 CPU 执行该进程代码的时间。程序是静态的，存放在硬盘上，是永久的，而进程是动态的，是暂时的，有其生命周期，当程序加载到内存时，操作系统为其分配好其进程空间时，从 main 函数开始运行时，标志着该进程生命的开始，当调用 exit 函数退出时，标志着该进程生命的结束，随后操作系统回收进程空间和所占资源。

每个 Linux 进程都一定有一个唯一的数字标识符，称为进程 ID (process ID)。进程 ID 总是一非负整数，使用 getpid 函数可得到本进程的进程 ID 号。

下面是 getpid 的函数范例，getpid.c 源代码如下：

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main()
{
    printf("pid=%d\n", getpid());
    printf("parent pid=%d\n", getppid());
    return 0;
}
```

编译 gcc getpid.c -o getpid。

执行 ./getpid，执行结果如下：

```
pid=6307
parent pid=5441
```

使用 ps -ef|grep 5441，得到如下结果：

```
zjkf      5441  5438  0 06:20 pts/0    00:00:00 -bash
```

其中 getpid.c 就是源程序，getpid 执行码就是编译后的可执行程序，./getpid 执行码执行的过程叫做进程。可见，不管源程序，还是编译后的可执行程序，都是静态的，而进程是动态的。getpid 进程的父进程为 Shell (bash) 进程，Shell 进程是所有在终端上执行进程的父进程。

3. 进程控制

用于进程控制的主要函数有 fork、exec (exec 函数有六种变体，但经常把它们统称为 exec 函数) 和 waitpid。

12.1.5 ANSI C

American National Standards Institute (ANSI——美国国家标准学会) 是由公司、

政府和其他成员组成的自愿组织。它们协商与标准有关的活动，审议美国国家标准，并努力提高美国在国际标准化组织中的地位。

由于美国在计算机早期发展中一直处于领先地位，因此 ANSI 的很多标准已经成为事实上的国际标准。其中常见的 ANSI ASCII 字符编码几乎为所有的编码方式所兼容。

标准的力量无处不在，标准是一种约定和约束，提供了技术目标说明与技术规范，标准的导向性和强制性加快了技术交流、提高了技术质量。

1. ANSI C

符合 ANSI 标准的 C 语言称为 ANSI C，现在的 C 语言程序一般都符合 ANSI C 标准。

2. ANSI C 函数原型

标准库函数的函数原型都在头文件中提供，程序可以用 `#include` 指令包含这些原型文件。对于用户自定义函数，程序员需要在头文件或文件头声明其原型。对于返回 `int` 型和 `void` 型且不带形参的 C 语言函数，可以不声明其函数原型，但不鼓励这么用。

函数原型描述了函数到编译器的接口，编译程序在编译时，就可以检查在调用函数时是否使用了正确的参数。

如 `getpid` 函数原型如下，其中 `pid_t` 是一个 `typedef` 定义类型，其定义为 `typedef int pid_t`。

```
pid_t  getpid ( void ) ;
```

调用带参数的 `getpid` (如 `getpid(1)`)，则 ANSI C 程序将给出下列形式的出错信息：

```
line 8: too many arguments to function " getpid "
```

另外，因为编译程序知道参数的数据类型，所以如果可能，它就会将参数强制转换成所需的数据类型。

3. ANSI C 类属指针

在标准 ANSI 中，`read` 和 `write` 的第二个参数现在是 `void *` 类型，而早期的 UNIX (Linux 是 UNIX 的后继者，站在 UNIX 的肩膀上) 系统都使用 `char *` 这种指针类型。

ANSI C 使用 `void *` 作为类属指针来代替 `char *`，这样函数原型和类属指针的组合消去了很多非 ANSI C 程序需要的显式类型强制转换。

例如，下面代码使用 ANSI C 标准，可以写成：

```
float data[100];  
write(fd,data,sizeof(data)) ;
```

若使用非 ANSI C 编译程序，则需写成：

```
write( fd , (void *)data , sizeof(data)) ;
```

12.1.6 用户标识

1. 用户 ID

在 Linux 系统，每个用户有一个唯一的用户 ID 号。

口令文件登录项中的用户 ID (user ID) 是个数值，它向系统标识各个不同的用户。系统管理员在确定一个用户登录名的同时，确定其用户 ID，用户不能更改其用户 ID，通常每个用户有一个唯一的用户 ID。

用户 ID 为 0 的用户为根 (root) 或超级用户 (superuser)。在口令文件中，通常有一个登录项，其登录名为 root，root 用户属于特权用户。如果一个进程具有超级用户特权，则大多数文件许可权检查都不再进行。某些操作系统功能只限于向超级用户提供，超级用户对系统有自由的支配权。

2. 组 ID

在 Linux 系统，每个用户有一个唯一的组 ID 号。

口令文件登录项也包括用户的组 ID (group ID)，它也是一个数值。组 ID 也是由系统管理员在确定用户登录名时分配的。

组文件将组名映射为数字组 ID，它通常是 /etc/group。

一个用户可以属于多个组，但只能属于一个主组，非主组又称为添加组或附属组。

getuid.c 源代码如下：

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main()
{
    printf("uid=%d, gid=%d\n", getuid(), getgid());
    return 0 ;
}
```

编译 gcc getuid.c -o getuid。

执行 ./getuid，执行结果如下：

```
uid=1008, gid=1003
```

12.1.7 出错处理

当 Linux 函数出错时，经常会给整型变量 errno 设置一个值，error 中每个值都表示特定的含义。

文件 <errno.h> 中定义了变量 errno 以及可以赋予它的各种常数宏定义，这些常数都以 E

开头，如 `errno` 等于常数 `EACCES` 时，表示“此进程没有打开该文件的权限”。

对于 `errno` 有两条规则：第一条规则是如果没有出错，则 `error` 的值不会被重设，因此，仅当函数的返回值指明出错时，才检验其值；第二条是任一函数都不会将 `errno` 值设置为 0，在 `<errno.h>` 中定义的所有常数都不为 0。

C 标准定义了两个函数，即 `strerror` 和 `perror`，它们帮助打印出错信息，这两个函数的函数原型如下：

```
#include <string.h>
char *strerror(int errnum);
```

此函数将 `errnum`（它通常就是 `errno` 值）映射为一个出错信息字符串，并且返回此字符串的指针。

`perror` 函数在标准出错上产生一条出错消息（基于 `errno` 当前值），然后返回。

```
#include <stdio.h>
void perror( const char *msg);
```

它首先输出由 `msg` 指向的字符串，然后是一个冒号、一个空格，然后是对应 `errno` 值的出错信息，最后是一个换行符。

下面以两个范例来说明 `strerror` 和 `perror` 的使用。

`strerror.c` 源代码如下，此代码打印出 0~131 的错误原因描述。

```
#include <stdio.h>
#include <string.h>
int main()
{
    int i;
    for(i=0;i<132;i++)
    {
        printf("%d : %s\n", i, strerror(i));
    }
    return 0 ;
}
```

编译 `gcc strerror.c -o strerror`。

执行 `./strerror`，执行结果如下：

```
0 : Success
1 : Operation not permitted
2 : No such file or directory
.....
```

`perror.c` 源代码如下，此函数打印出错误原因信息字符串。

```
#include <stdio.h>
int main()
{
    FILE *fp;
```

```
fp = fopen("/tmp/noexist", "r+") ;
if ( fp == NULL )
{
    perror("fopen") ;
    return -1 ;
}
return 0 ;
}
```

编译 `gcc peror.c -o perror`。

执行 `./perror`，执行结果如下：

```
fopen: No such file or directory
```

12.1.8 Linux 信号、时间值与系统调用

1. Linux 信号

信号是通知进程已发生某种条件的一种技术。例如，若某一进程执行除法操作，其除数为 0，则将名为 SIGFPE 的信号发送给该进程。进程如何处理信号有三种选择：

① 忽略该信号。有些信号不能忽略，如除以 0 或访问进程地址空间以外的单元等，这些异常产生的后果不确定。

② 按系统默认方式处理。对于除 0，系统默认方式是终止该进程。

③ 提供一个函数，信号发生时则调用该函数。使用这种方式，能知道什么时候产生了信号，并按所希望的方式处理它。

中断键（通常是 Delete 键或 Ctrl+C）和退出键（通常是 Ctrl+\）会产生信号，它们被用于中断当前的运行进程。调用 kill 函数也会产生信号，在一个进程中调用 kill 函数就可向其他进程发送一个信号，但必须有其用户权限。

2. Linux 时间值

Linux 有两种不同的时间值，分别为日历时间和进程时间。

（1）日历时间。该值是自 1970 年 1 月 1 日 00:00:00 以来国际标准时间（UTC）所经过的秒数累计值（UTC 为格林尼治标准时间），这些时间值可用于记录文件最近一次的修改时间等。

（2）进程时间。这也称为 CPU 时间，用以度量进程使用 CPU 资源。

当度量一个进程的执行时间时，Linux 系统使用三个进程时间值：时钟时间、用户 CPU 时间和系统 CPU 时间。

时钟时间又称为墙上时钟时间（wall clock time），它是进程运行的时间总量，其值与系统中同时运行的进程数有关。

用户 CPU 时间是执行用户指令所用的时间量。

系统 CPU 时间是为该进程执行内核所经历的时间。

使用 `time ./执行码` 可以获得该进程的时钟时间、用户时间和系统时间。

```
$time ./test
real    0m0.06s
user    0m0.01s
sys     0m0.00s
```

3. 系统调用和库函数

系统调用和库函数相关概念如下。

- ① 系统调用：系统调用把应用程序的请求传给内核，调用相应的内核函数完成所需的处理，将处理结果返回给应用程序。系统调用使应用程序由用户态进入核心态，系统调用有自己单独的堆栈空间，系统调用也是以函数的形式提供给应用程序使用。
- ② 库函数：存放在函数库中的函数，库函数具有明确的功能、入口调用参数和返回值，Linux 中库函数特指函数入口没有进行系统调用的库函数。库函数中常包含系统调用，库函数没有进行系统调用时，没有单独的堆栈空间。
- ③ 用户态：进程的普通执行状态，在用户态下进程具有较低的特权，只能执行规定的机器指令，不能执行特权指令。进程在用户态下只能访问该进程的存储空间，不能与系统硬件相互作用，不能访问系统资源，当它需要系统硬件资源时，会通过系统调用进入核心态。
- ④ 核心态：核心态，或者特权态（与之相对应的是用户态），能执行所有的机器指令，包括由操作系统执行的特权指令，能访问所有的寄存器和存储区域，能直接控制所有的系统资源和硬件资源。Linux 在执行内核程序时，处于核心态。

图 12-2 给出了系统调用与库函数原理图，用户应用代码可以直接调用库函数或者系统调用，库函数属于用户空间，而系统调用属于系统空间。

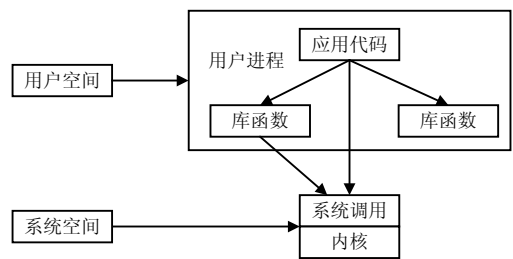


图 12-2 系统调用与库函数

从执行者的角度来看，系统调用和库函数之间有重大区别，但从用户角度来看，其区别并不非常重要。

库函数可能会调用一个或多个内核的系统调用，但是它们并不是内核的入口点，例如，`printf` 函数会调用 `write` 系统调用以进行输出操作。库函数也可以不调用任何系统调用，如函数 `strcpy` 和 `atoi` 并不使用任何系统调用。

4. Linux 系统调用函数列表

系统调用是操作系统提供给应用程序访问硬件资源和进行操作系统内核操作的接口。表 12-1 列出了 Linux 系统调用中用到的函数，共分为进程控制、文件系统控制、系统管理、内存管理、网络管理、socket 控制、用户管理、进程间通信八个部分。

表 12-1 Linux 系统调用函数列表

函 数 名	含 义
1. 进程控制	
fork	创建一个新进程
clone	按指定条件创建子进程
execve	运行可执行文件
exit	中止进程
_exit	立即中止当前进程
getdtablesize	进程所能打开的最大文件数
getpgid	获取指定进程组标识号
setpgid	设置指定进程组标识号
getpgrp	获取当前进程组标识号
setpgrp	设置当前进程组标识号
getpid	获取进程标识号
getppid	获取父进程标识号
getpriority	获取调度优先级
setpriority	设置调度优先级
modify_ldt	读写进程的本地描述表
nanosleep	使进程睡眠指定的时间
nice	改变分时进程的优先级
pause	挂起进程，等待信号
personality	设置进程运行域
prctl	对进程进行特定操作
ptrace	进程跟踪
sched_get_priority_max	取得静态优先级的上限
sched_get_priority_min	取得静态优先级的下限
sched_getparam	取得进程的调度参数
sched_getscheduler	取得指定进程的调度策略
sched_rr_get_interval	取得按 RR 算法调度的实时进程的时间片长度
sched_setparam	设置进程的调度参数
sched_setscheduler	设置指定进程的调度策略和参数
sched_yield	进程主动让出处理器，并将自己置于等候调度队列队尾
vfork	创建一个子进程，以供执行新程序，常与 execve 等同时使用
wait	等待子进程终止
wait3	参见 wait 的含义
waitpid	等待指定子进程终止

续表

函 数 名	含 义
wait4	参见 waitpid 的含义
capget	获取进程权限
capset	设置进程权限
getsid	获取会话标识号
setsid	设置会话标识号
2. 文件系统控制	
(1) 文件读写操作	
fcntl	文件控制
open	打开文件
creat	创建新文件
close	关闭文件描述字
read	读文件
write	写文件
readv	从文件读入数据到缓冲数组中
writew	将缓冲数组里的数据写入文件
pread	对文件随机读
_llseek	在 64 位地址空间里移动文件指针
dup	复制已打开的文件描述字
dup2	按指定条件复制文件描述字
flock	文件加/解锁
poll	I/O 多路转换
truncate	截断文件
ftruncate	参见 truncate 的含义
umask	设置文件权限掩码
fsync	把文件在内存中的部分写回磁盘
(2) 文件系统操作	
access	确定文件的可存取性
chdir	改变当前工作目录
fchdir	参见 chdir 的含义
chmod	改变文件方式
fchmod	参见 chmod 的含义
chown	改变文件的属主或用户组
fchown	参见 chown 的含义
lchown	参见 chown 的含义
chroot	改变根目录
stat	取文件状态信息
lstat	参见 stat 的含义
fstat	参见 stat 的含义
statfs	取文件系统信息
fstatfs	参见 statfs 的含义

续表

函 数 名	含 义
readdir	从 DIR 目录流结构中每次读取一个目录项
getdents	一次读取所有目录项
mkdir	创建目录
mknod	创建索引节点
rmdir	删除目录
rename	文件改名
link	创建链接
symlink	创建符号链接
unlink	删除链接
readlink	读符号链接的值
mount	安装文件系统
umount	卸下文件系统
ustat	取文件系统信息
utime	改变文件的访问修改时间
utimes	参见 utime 的含义
quotactl	控制磁盘配额
3. 系统管理	
ioctl	I/O 总控制函数
_sysctl	读/写系统参数
acct	启用或禁止进程记账
getrlimit	获取系统资源上限
setrlimit	设置系统资源上限
getrusage	获取系统资源使用情况
uselib	选择要使用的二进制函数库
ioperm	设置端口 I/O 权限
iopl	改变进程 I/O 权限级别
outb	低级端口操作
reboot	重新启动
swapon	打开交换文件和设备
swapoff	关闭交换文件和设备
bdflush	控制 bdflush 守护进程
sysfs	取核心支持的文件系统类型
sysinfo	取得系统信息
adjtimex	调整系统时钟
alarm	设置进程的闹钟
getitimer	获取计时器值
setitimer	设置计时器值
gettimeofday	取时间和时区
settimeofday	设置时间和时区
stime	设置系统日期和时间

续表

函 数 名	含 义
time	取得系统时间
times	取进程运行时间
uname	获取当前 Linux 系统的名称、版本和主机等信息
vhangup	挂起当前终端
nfsservctl	对 NFS 守护进程进行控制
vm86	进入模拟 8086 模式
create_module	创建可装载的模块项
delete_module	删除可装载的模块项
init_module	初始化模块
query_module	查询模块信息
*get_kernel_syms	取得核心符号，已被 query_module 代替
4. 内存管理	
brk	改变数据段空间的分配
sbrk	参见 brk
mlock	内存页面加锁
munlock	内存页面解锁
mlockall	调用进程所有内存页面加锁
munlockall	调用进程所有内存页面解锁
mmap	映射虚拟内存页
munmap	去除内存页映射
mremap	重新映射虚拟内存地址
msync	将映射内存中的数据写回磁盘
mprotect	设置内存映像保护
getpagesize	获取页面大小
sync	将内存缓冲区数据写回硬盘
cacheflush	将指定缓冲区中的内容写回磁盘
5. 网络管理	
getdomainname	取域名
setdomainname	设置域名
gethostid	获取主机标识号
sethostid	设置主机标识号
gethostname	获取本主机名称
sethostname	设置主机名称
6. socket 控制	
socketcall	socket 系统调用
socket	建立 socket
bind	绑定 socket 到端口
connect	连接远程主机
accept	响应 socket 连接请求
send	通过 socket 发送信息

续表

函 数 名	含 义
sendto	发送 UDP 信息
sendmsg	参见 send 的含义
recv	通过 socket 接收信息
recvfrom	接收 UDP 信息
recvmsg	参见 recv 的含义
listen	监听 socket 端口
select	对多路同步 I/O 进行轮询
shutdown	关闭 socket 上的连接
getsockname	取得本地 socket 名字
getpeername	获取通信对方的 socket 名字
getsockopt	取端口设置
setsockopt	设置端口参数
sendfile	在文件或端口间传输数据
socketpair	创建一对已连接的无名 socket
7. 用户管理	
getuid	获取用户标识号
setuid	设置用户标识号
getgid	获取组标识号
setgid	设置组标识号
getegid	获取有效组标识号
setegid	设置有效组标识号
geteuid	获取有效用户标识号
seteuid	设置有效用户标识号
setregid	分别设置真实和有效的组标识号
setreuid	分别设置真实和有效的用户标识号
getresgid	分别获取真实的，有效的和保存过的组标识号
setresgid	分别设置真实的，有效的和保存过的组标识号
getresuid	分别获取真实的，有效的和保存过的用户标识号
setresuid	分别设置真实的，有效的和保存过的用户标识号
setfsgid	设置文件系统检查时使用的组标识号
setfuid	设置文件系统检查时使用的用户标识号
getgroups	获取后补组标识清单
setgroups	设置后补组标识清单
8. 进程间通信	
(1) 信号	
sigaction	设置对指定信号的处理方法
sigprocmask	根据参数对信号集中的信号执行阻塞/解除阻塞等操作
sigpending	为指定的被阻塞信号设置队列
sigsuspend	挂起进程等待特定信号
signal	参见 signal

续表

函 数 名	含 义
kill	向进程或进程组发信号
sigblock	向被阻塞信号掩码中添加信号，已被 sigprocmask 代替
siggetmask	取得现有阻塞信号掩码，已被 sigprocmask 代替
sigsetmask	用给定信号掩码替换现有阻塞信号掩码，已被 sigprocmask 代替
sigmask	将给定的信号转化为掩码，已被 sigprocmask 代替
sigpause	作用同 sigsuspend，已被 sigsuspend 代替
sigvec	为兼容 BSD 而设的信号处理函数，作用类似 sigaction
ssetmask	ANSI C 的信号处理函数，作用类似 sigaction
(2) 消息	
msgctl	消息控制操作
msgget	获取消息队列
msgsnd	发消息
msgrcv	取消息
(3) 管道	
pipe	创建管道
(4) 信号量	
semctl	信号量控制
semget	获取一组信号量
semop	信号量操作
(5) 共享内存	
shmctl	控制共享内存
shmget	获取共享内存
shmat	连接共享内存
shmdt	断开共享内存

12.2 Linux 进程环境

本节将介绍下面几个方面的内容：Linux 进程控制块的作用；与进程有关的 ID；当执行程序时，其 main 函数是如何被调用的；命令行参数是如何传送给执行程序的；典型的存储器布局是什么样式；如何动态分配存储空间；进程如何使用环境变量；进程终止的不同方式；全局跳转函数 setjmp 和 longjmp 的使用。

1. Linux 进程控制块

进程由进程控制块（PCB）和进程堆栈两部分组成。进程控制块由操作系统管理，实现的是进程的管理和控制功能；进程堆栈对应着程序，实现的是程序的执行功能。

进程控制块（PCB）是操作系统为了管理进程设置的一个专门的数据结构，用它来记录进程的外部特征，描述进程的运动变化过程。操作系统利用 PCB 来控制和管理进程，所以 PCB 是操作系统感知进程存在的唯一标志，进程与 PCB 是一一对应的。Linux 进程控制块是一个由结构

task_struct 所定义的数据结构。由于进程运行态分为运行态、就绪态和等待态，单 CPU 时，运行态进程只有一个，而就绪态和等待态的进程有多个，操作系统维护就绪态进程和等待态进程是通过两个队列链表完成的。图 12-3 给出了 Linux 进程等待队列图，等待队列是通过循环链表实现的。

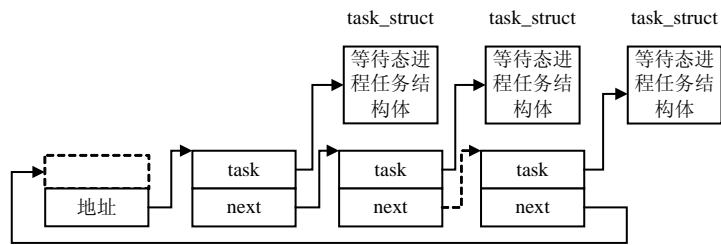


图 12-3 Linux 进程等待队列图

图 12-4 给出了 Linux 进程控制块 task_struct 的结构图，task_struct 结构中记录了单个进程的管理信息，其中 files 指向的是文件列表。

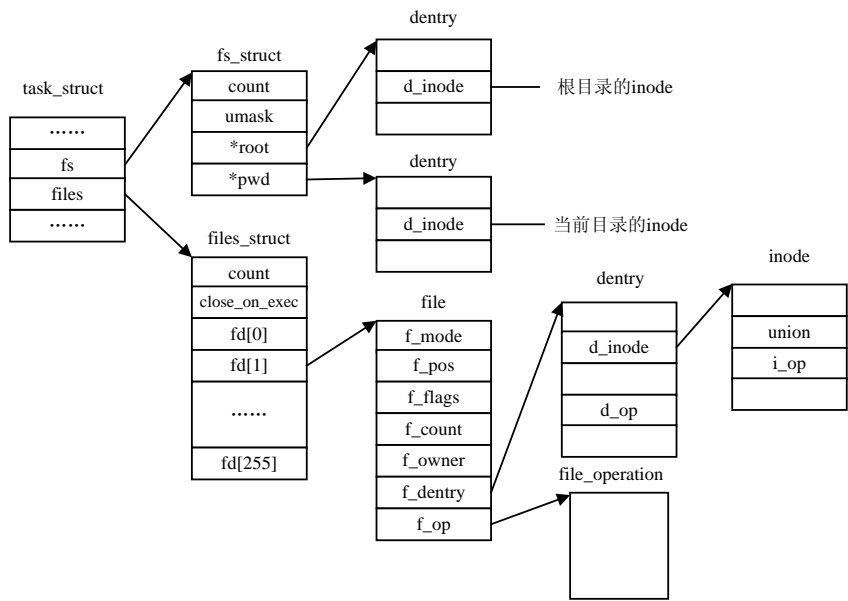


图 12-4 进程控制块 task_struct 结构图

2. 与进程有关的 ID

与进程有关的 ID 主要有下面六种，这六种 ID 是 Linux 进程环境经常用到的 ID 号。

- ① 真正用户标识号（UID）：该标识号标识运行进程属于哪一个用户。
- ② 有效用户标识号（EUID）：给新创建的进程赋予的用户权限。
- ③ 真正用户组标识号（GID）：该标识号标识运行进程属于哪一个用户组。
- ④ 有效用户组标识号（EGID）：给新创建的进程赋予的用户组权限。
- ⑤ 进程标识号（PID）：用数字来标识进程 ID 号。

⑥ 进程组标识号 (process group ID): 标识该进程属于哪一个进程组。

3. main 函数

C 程序总是从 main 函数开始执行, main 函数的原型是:

```
int main( int argc, char *argv[]);
```

其中, argc 是命令行参数的数目, argv 为指针数组, 存放 main 函数传入的参数。当内核启动 C 程序时 (使用一个 exec 函数), 在调用 main 前先调用一个特殊的启动例程。可执行程序文件将此启动例程指定为程序的起始地址, 这是由 C 语言编译链接程序安排的。启动例程从内核取得命令行参数和环境变量值, 然后为调用 main 函数做好安排。

4. 进程终止

在 Linux 环境下, 有五种方式使进程终止, 其说明如下。

(1) 正常终止

- ① 从 main 返回。
- ② 调用 exit。
- ③ 调用 _exit。

(2) 异常终止

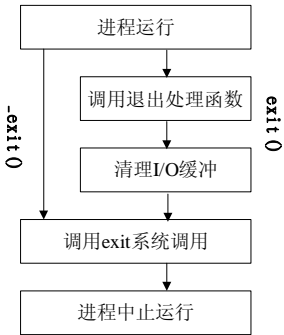
- ① 调用 abort。
- ② 由一个信号终止。

5. exit 和 _exit 函数

(1) exit 和 _exit 函数说明

exit 和 _exit 函数都是用来终止进程的。当程序执行到 exit 或 _exit 时, 进程会无条件地停止剩下的所有操作, 清除包括 PCB 在内的各种数据结构, 并终止本进程的运行。但是, 这两个函数还是有区别的, 如图 12-5 所示。

从图 12-5 中可以看出, _exit 函数的作用是: 直接使进程停止运行, 清除其使用的内存空间, 并清除其在内核中的各种数据结构。exit 函数则在这些基础上做了一些包装, 在执行退出之前加了若干道工序。exit 函数与 _exit 函数最大的区别就在于 exit 函数在退出之前要检查文件的打开情况, 把文件缓冲区中的内容写回文件, 就是图中的“清理 I/O 缓冲”一项。



由于在 Linux 的标准函数库中, 有一种被称为“缓冲 I/O” 图 12-5 exit 与 _exit 函数原理比较

(buffered I/O)”的操作，其特征就是对应每一个打开的文件，在内存中都有一片缓冲区。每次读文件时，会连续读出若干条记录，这样在下次读文件时，就可以直接从内存的缓冲区中读取。同样，每次写文件的时候，也仅仅是写入内存中的缓冲区，等满足了一定的条件（如达到一定数量或遇到特定字符等），再将缓冲区中的内容一次性写入文件。

这种技术大大增加了文件读写速度，但也为编程带来了一点麻烦，比如有一些数据，认为已经写入了文件，实际上因为没有满足特定的条件，它们还只是保存在缓冲区内，这时用_exit函数直接将进程关闭，缓冲区中的数据就会丢失。因此，若想保证数据的完整性，就一定要使用exit函数。

(2) exit 和_exit 函数语法

exit 用来正常结束进程，其函数原型说明如下。

exit（正常结束进程）	
所需头文件	#include <stdlib.h>
函数说明	exit()用来正常终止目前进程的执行，并把参数 status 返回给父进程，而进程所有的缓冲区数据会自动写回并关闭未关闭的文件
函数原型	void exit(int status)
函数传入值	status: status 是一个整型的参数，可以利用这个参数传递进程结束时的状态。一般来说，0 表示正常结束，其他的数值表示出现了错误，进程非正常结束。在实际编程时，可以用 wait 系统调用接收子进程的返回值，从而针对不同的情况进行不同的处理

_exit 用来结束进程执行，其函数原型说明如下。

_exit（立刻结束进程）	
所需头文件	#include <stdlib.h>
函数说明	_exit()用来立刻结束目前进程的执行，并把参数 status 返回给父进程，并关闭未关闭的文件。此函数调用后不会返回，并且会传递 SIGCHLD 信号给父进程，父进程可以由 wait 函数取得子进程结束状态
函数原型	void _exit(int status)
函数传入值	status: 同 exit 函数含义
附加说明	_exit()不会处理标准 I/O 缓冲区，如要更新缓冲区请使用 exit()

(3) exit 函数处理流程图

exit 函数首先调用各终止处理程序，然后按需多次调用 fclose，关闭所有打开流。图 12-6 显示了一个 C 程序是如何启动的，以及它终止的各种方式。

(4) exit 函数举例

exit.c 源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    int i ;
```

```
printf("count of arguments:%d\n", argc); /*打印命令行参数的个数*/
for( i=0; i<argc; i++)
    printf("argv[%d]:%s\n", i, argv[i]); /*逐个打印命令行参数*/
printf("Using exit...\n");
printf("This is the content in buffer");
exit(3);
}
```

编译 gcc exit.c -o exit。

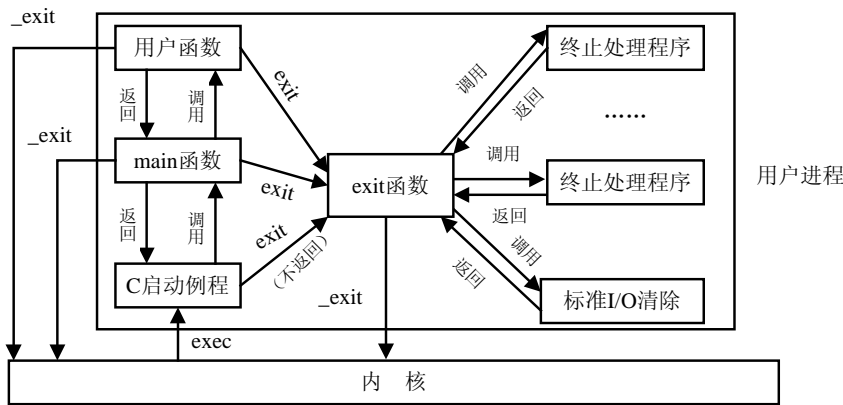


图 12-6 exit 函数调用图

执行 `./exit arg1 arg2`，结果如下：

```
argv[0]:./exit
argv[1]:arg1
argv[2]:arg2
Using exit...
This is the content in buffer
```

从输出的结果中可以看到，调用 `exit` 函数时，缓冲区（“This is the content in buffer”）中的内容也能正常输出。该行没有 `\n` 换行符，所写内容在内存中，可见利用 `exit` 函数退出前刷新了内存。

执行 `echo $?` 得到该进程的退出状态如下：

```
3
```

(5) `_exit` 函数举例

`_exit.c` 源代码如下：

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
    printf("Using _exit...\n");
    printf("This is the content in buffer");
}
```

```
// fflush(stdout);
_exit(1);
}
```

编译 gcc _exit.c -o _exit。

执行./_exit，执行结果如下：

```
Using _exit...
```

从最后的结果中可以看到，调用_exit 函数无法输出缓冲区中的记录。因为_exit 函数不刷新文件描述符的缓存，使用_exit 函数前可以用 fflush 函数把缓存内容刷新到标准输出上。

(6) atexit 函数说明

按照 ANSI C 的规定，一个进程可以登记多达 32 个函数，这些函数将由 exit 自动调用，这些函数为终止处理程序 (exit handler)，并用 atexit 函数来登记这些函数。操作系统把 atexit 的注册函数依次压栈，exit 函数执行前将先处理 atexit 函数注册的函数。

atexit 是设置程序正常结束前调用的函数，其函数原型如下。

atexit（登记终止处理程序）	
所需头文件	#include <stdlib.h>
函数说明	atexit()用来设置一个程序正常结束前调用的函数。当程序通过调用 exit()或从 main 中返回时，参数 function 所指定的函数会先被调用，然后才真正由 exit()结束程序
函数原型	int atexit (void (*function)(void))
函数返回值	如果执行成功则返回 0，否则返回-1，失败原因存于 errno 中

atexit 函数举例，atexit.c 源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
void my_exit(void)
{
    printf("before exit () !\n");
}
int main()
{
    atexit(my_exit);
    printf("test test\n") ;
    exit(0);
}
```

编译 gcc atexit.c -o atexit。

执行./atexit，执行结果如下：

```
test test
before exit () !
```

(7) 得到进程结束状态

Linux 程序员可以通过 Shell 得到已结束进程的结束状态：

```
$执行码
$echo $?
```

\$?是 Linux Shell 中一个内置变量，其中保存的是最近一次运行的进程的返回值，这个返回值有以下 3 种情况：

- ① 程序中的 main 函数运行结束，\$?中保存 main 函数的返回值。
- ② 程序运行中调用 exit 函数结束运行，\$?中保存 exit 函数的参数。
- ③ 程序异常退出，\$?中保存异常出错的错误号。

6. 进程的堆栈空间

每一个进程都有自己的一个进程堆栈空间。在 Linux 界面执行一个执行码时，Shell 进程会 fork 一个子进程，再调用 exec 系统调用在子进程中执行该执行码。

exec系统调用执行新程序时，会把命令行参数和环境变量表传递给main函数，它们在整个进程堆栈空间中的位置如图 12-7 所示。

下面是对图 12-7 进程堆栈空间各段的具体说明：

- ① 代码段（文本段）：保存程序的执行码。在进程并发时，代码段是共享的且只读的，在存储器中只需有一个副本。
- ② 数据段：此段又称为初始化数据段，它包含了程序中已初始化的全局变量、全局静态变量、局部静态变量。例如，在函数外定义变量并赋值：`int count=30 ;`，此变量 count 存放在数据段中。

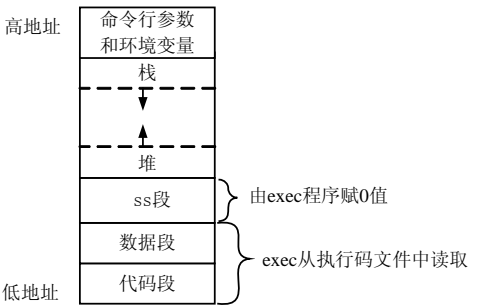


图 12-7 进程堆栈空间图

- ③ bss 段：通常此段又称为未初始化数据段，它包含了程序中未初始化的全局变量、全局静态变量、局部静态变量，程序执行前操作系统将此段初始化为 0。例如，在函数外定义了变量但没有赋值：`long sum[1000] ;`，此变量存放在 bss 段中。

- ④ 栈：程序执行前静态分配的内存空间，栈的大小可在编译时指定，Linux 环境下默认为 8M。栈段是存放程序执行时局部变量、函数调用信息、中断现场保留信息的空间。程序执行时，CPU 堆栈段指针会在栈顶根据执行情况进行上下移动。

- ⑤ 堆：程序执行时，按照程序需要动态分配的内存空间。malloc、calloc、realloc 函数都在堆上分配空间。

7. 环境变量

环境变量用于描述该用户操作环境下特定意义的变量，可以用 `env` 命令查看该用户下生效的环境变量。

和命令行参数 `argv` 类似，环境变量表也是一组字符串，如图 12-8 所示。

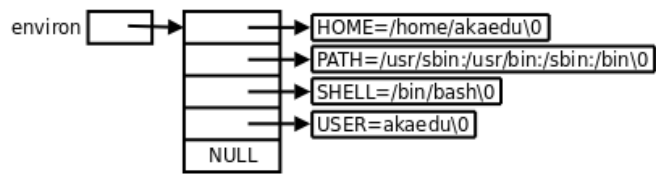


图 12-8 环境变量图

`libc` 库中定义的全局变量 `environ` 指向环境变量表，由于 `environ` 变量没有包含在任何头文件中，所以在使用时要用 `extern` 声明。

依照惯例，环境变量字符串都是以 `name=value` 这样的形式保存在内存中的。大多数 `name` 由大写字母加下画线组成，一般把 `name` 部分叫做环境变量名，`value` 部分则是环境变量的值，而且 `value` 需要以 `'\0'` 结尾，环境变量定义了该进程的运行环境。

8. 打印环境变量

下面是打印环境变量的程序范例。

`environ.c` 源代码如下：

```
#include <stdio.h>
int main(void)
{
    extern char **environ;
    int i;
    for(i=0; environ[i]!=NULL; i++)
        printf("%s\n", environ[i]);
    return 0;
}
```

编译 `gcc environ.c -o environ`。

执行 `./environ`，执行结果如下：

```
HOME=/home/zjkf
LANGUAGE=zh_CN:zh:en_US:en
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
.....
```

在命令行执行时，`Shell` 进程是执行进程的父进程。由于父进程在调用 `fork` 创建子进程后再调用 `exec` 函数执行 `environ` 程序，`Shell` 进程会把自身的环境变量表复制给子进程，所以 `./environ` 打印的环境变量和 `Shell` 进程的环境变量是相同的。

9. 环境变量函数

(1) 环境变量函数

环境变量的使用会用到 `getenv`、`setenv`、`unsetenv` 这三个函数，这三个函数的原型及说明如下。

getenv（得到环境变量）	
所需头文件	#include <stdlib.h>
函数说明	getenv()用来取得参数 name 环境变量的内容。参数 name 为环境变量的名称，如果该变量存在则会返回指向该内容的指针
函数原型	char * getenv(const char *name)
函数返回值	成功：返回指向该内容的指针
	出错：NULL

setenv（改变或增加环境变量）	
所需头文件	#include <stdlib.h>
函数说明	setenv()用来改变或增加环境变量的内容。参数 name 为环境变量名称字符串
函数原型	int setenv(const char *name,const char * value,int overwrite)
函数传入值	name: 环境变量名称
	value: 环境变量的值
	overwrite:
	■ 0 表示该环境变量存在时，value 则忽略 ■ 非 0 表示该环境变量存在时，原内容会被改为参数 value 所指的变量内容
函数返回值	成功：返回指向该内容的指针
	出错：-1，错误原因存于 errno 中
错误代码	ENOMEM: 内存不足，无法配置新的环境变量空间

unsetenv（删除环境变量）	
所需头文件	#include <stdlib.h>
函数说明	unsetenv()用来删除环境变量
函数原型	void unsetenv(const char *name)
函数传入值	name: 环境变量名称

(2) 环境变量函数使用示例

`getenv.c` 源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char * p;
    if((p=getenv("USER"))){
        printf("USER =%s\n",p);
        setenv("USER","test",1);
    }
```

```
printf("USER=%s\n",getenv("USER"));
unsetenv("USER");
printf("USER=%s\n",getenv("USER"));

return 0 ;
}
```

编译 gcc getenv.c -o getenv。

执行 ./getenv，执行结果如下：

```
USER =zjkf
USER=test
USER=(null)
```

(3) 环境变量经常使用的场合

环境变量常使用在得到一个文件的全路径名的场合。使用时需要先在用户的.profile 中设置 FILEDIR=\$HOME/print/;export FILEDIR,然后在程序中使用 sprintf(file_full_name , "%s%s",getenv("FILEDIR") ,filename) 语句，就可以得到文件的全路径名称，其中 filename 是文件名，file_full_name 用来存放文件的全路径名称。

10. 内存分配函数

(1) 内存分配函数原型

常用的内存分配函数有 malloc、calloc 和 realloc，其中，calloc 分配后的数据会被设置为 0，malloc 分配的数据不会被初始化，realloc 新分配的区域没有被初始化。这三个函数申请的内存空间需要用 free 来释放。其中，malloc 函数已在 C 语言章节中介绍过，calloc 和 realloc 函数的函数原型说明如下。

calloc（申请内存空间）	
所需头文件	#include <stdlib.h>
函数说明	申请内存空间，返回申请内存空间的首地址
函数原型	void *calloc(size_t nmemb, size_t size)
函数传入值	nmemb: 申请内存单位的个数
	size: 申请内存单位的大小
函数返回值	申请内存单位的首地址

realloc（申请一片内存空间）	
所需头文件	#include <stdlib.h>
函数说明	申请内存空间，返回申请内存空间的首地址
函数原型	void *realloc(void *ptr, size_t size)
函数传入值	ptr: 先前申请的内存地址
	size: 新申请内存空间的大小
函数返回值	申请内存空间的地址

续表

realloc（申请一片内存空间）	
附加说明	<div>① 若新申请的空间比原来小，内存内容不变，返回的仍是原先申请的首地址</div> <div>② 若新申请的空间比原来大，原先的内存空间后面还有足够的空闲空间用来分片，那么扩展原来申请的空间，原有内容保持不变，返回的仍是原先申请的首地址</div> <div>③ 若新申请的空间比原来大，原先的内存空间后面没有足够内存进行分片，则另外申请一片大小为 size 的空间，并把原内存空间中的内容复制过来，并且释放原先的内存空间，返回新申请的内存空间地址</div>

（2）内存分配函数使用实例

memalloc.c 源代码如下：

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
struct test
{
    int a[10];
    char b[20];
};
int main()
{
    struct test *ptr0=(struct test *)calloc(sizeof(struct test),10);
    struct test *ptr1=(struct test *)malloc(sizeof(struct test)*10);
    memset(ptr1, 0x00, sizeof(struct test)*10) ;
    free(ptr0) ;
    free(ptr1) ;
    return 0;
}
```

11. 函数间跳转

在 C 语言中，goto 语句只允许在函数内使用，不允许在函数间使用。而执行这种跳转功能的是函数 setjmp 和 longjmp，这两个函数对于处理很深的嵌套函数调用中的出错问题非常有用。

（1）setjmp 函数原型

setjmp（保存目前堆栈环境）	
所需头文件	#include <setjmp.h>
函数说明	setjmp 用来保存堆栈环境，然后将目前的地址做一个记号，在程序其他地方调用 longjmp 时，便会直接跳 到这个记号位置。然后还原堆栈环境，继续程序执行。setjmp 首次返回 0，代表已做好记号，若返回非 0 值，代表由 longjmp() 跳转回来
函数原型	int setjmp(jmp_buf env)
函数传入参数	env: 保存堆栈环境，要求声明为全局变量
函数返回值	返回 0 代表已保存好目前堆栈环境，随时可供 longjmp() 调用，若返回非 0 值，则代表是由 longjmp() 返回

（2）longjmp 函数原型

longjmp (跳转到原先 setjmp 保存的堆栈环境)	
所需头文件	#include <setjmp.h>
函数说明	longjmp 会还原到由 setjmp 保存的堆栈环境，然后跳转到 setjmp()之后继续程序的流程
函数原型	void longjmp(jmp_buf env,int val)
函数传入参数	env: setjmp 所保存的堆栈环境
	val: 是提供 setjmp()的返回值，此值不可为 0，若为 0，系统自动以 1 代替
函数返回值	无

(3) setjmp 与 longjmp 函数使用范例

longjmp.c 源代码如下:

```
#include <stdio.h>
#include <setjmp.h>
jmp_buf env ;
int test()
{
    printf("Before longjmp()\n") ;
    longjmp(env, 123456) ;
    printf("After longjmp()\n") ;
    return 0 ;
}
int main()
{
    int val ;
    int i = 1234 ;
    if ( ( val = setjmp( env ) ) != 0 )
    {
        printf("longjmp call!\n") ;
        printf("val=%d, i=%d\n", val, i ) ;
        return 0 ;
    }
    i = 5678 ;
    test() ;

    return 0 ;
}
```

编译 gcc longjmp.c -o longjmp。

执行 ./longjmp，执行结果如下:

```
Before longjmp()
longjmp call!
val=123456, i=5678
```

12.3Linux 进程控制

1. 进程标识

(1) 进程标识说明

每个进程都有一个非负整型的唯一进程 ID。因为进程 ID 标识符总是唯一的，常将其用做其

他标识符的一部分以保证其唯一性。

在 Linux 中，进程 ID 0 是调度进程，常常被称为交换进程。该进程并不执行任何磁盘上的程序——它是内核的一部分，因此也被称为系统进程。进程 ID 1 通常是 init 进程，在自举过程结束时，由内核调用。

init 通常读与系统有关的初始化文件（/etc/rc*文件），并将系统引导到一个状态（如多用户），init 进程决不会终止，它是一个普通的用户进程（与交换进程不同，它不是内核中的系统进程），但是它以超级用户特权态运行，init 进程是所有 儿进程的父进程。

在 Linux 中最主要的进程标识有进程号（PID，Process Identity Number）和它的父进程号（PPID，parent process ID），其中，PID 唯一地标识一个进程，PID 和 PPID 都是非零的正整数。

（2）进程标识相关函数原型

取得进程标识函数		
头文件	#include <sys/types.h> #include <unistd.h>	
函数说明	函数原型	返回值
	pid_t getpid(void)	进程 ID
	pid_t getppid(void)	父进程 ID
	uid_t getuid(void)	实际用户 ID
	uid_t geteuid(void)	有效用户 ID
	gid_t getgid(void)	实际组 ID
	gid_t getegid(void)	有效组 ID
	struct passwd * getpwuid(uid_t uid)	用来逐一搜索参数 uid 指定的用户识别码，找到时便将 该用户的信息以 passwd 结构返回

（3）进程标识函数使用实例

pid.c 源代码如下：

```
#include <unistd.h>
#include <pwd.h>
#include <sys/types.h>
#include <stdio.h>
int main(int argc, char **argv)
{
    pid_t my_pid, parent_pid;
    uid_t my_uid, my_euid;
    gid_t my_gid, my_egid;
    struct passwd *my_info;
    my_pid=getpid();
    parent_pid=getppid();
    my_uid=getuid();
    my_euid=geteuid();
    my_gid=getgid();
    my_egid=getegid();
```

```

my_info=getpwuid(my_uid);
printf("Process ID%ld\n",my_pid);
printf("Parent ID%ld\n",parent_pid);
printf("User ID%ld\n",my_uid);
printf("Effective User ID%ld\n",my_euid);
printf("Group ID%ld\n",my_gid);
printf("Effective Group ID%ld\n",my_egid) ;
    if(my_info)
    {
        printf("My Login Name%s\n" ,my_info->pw_name);
        printf("My Password %s\n" ,my_info->pw_passwd);
        printf("My User ID %ld\n",my_info->pw_uid);
        printf("My Group ID %ld\n",my_info->pw_gid);
        printf("My Real Name %s\n" ,my_info->pw_gecos);
        printf("My Home Dir %s\n", my_info->pw_dir);
        printf("My Work Shell%s\n", my_info->pw_shell);
    }
}

```

编译 `gcc pid.c -o pid`。

执行 `./pid`，执行结果如下：

```

Process ID6492
Parent ID5707
User ID1008
Effective User ID1008
Group ID1003
Effective Group ID1003
My Login Name zjkf
My Password x
My User ID 1008
My Group ID 1003
My Real Namezjkf,,,
My Home Dir /home/zjkf
My Work Shell/bin/bash

```

2. fork 系统调用

(1) fork 系统调用说明

`fork` 系统调用用于从已存在进程中创建一个新进程，新进程称为子进程，而原进程称为父进程。`fork` 调用一次，返回两次，这两个返回分别带回它们各自的返回值，其中在父进程中的返回值是子进程的进程号，而子进程中的返回值则是 0。因此，可以通过返回值来判定该进程是父进程还是子进程。

使用 `fork` 函数得到的子进程是父进程的一个复制品，它从父进程处继承了整个进程的地址空间，包括进程上下文、进程堆栈、内存信息、打开的文件描述符、信号控制设定、进程优先级、进程组号、当前工作目录、根目录、资源限制、控制终端等，而子进程所独有的只有它的进程号、计时器等。因此可以看出，使用 `fork` 系统调用的代价是很大的，它复制了父进程中的数据段和堆栈段里的绝大部分内容，使得 `fork` 系统调用的执行速度并不很快。

`fork` 的返回值这样设计是有原因的, `fork` 在子进程中返回 0, 子进程仍可以调用 `getpid` 函数得到自己的进程 ID, 也可以调用 `getppid` 函数得到父进程的进程 ID。在父进程中使用 `getpid` 函数可以得到自己的进程 ID, 然而要想得到子进程的进程 ID, 只有将 `fork` 的返回值记录下来, 别无他法。

`fork` 的另一个特性是所有由父进程打开的文件描述符都被复制到子进程中。父、子进程中相同编号的文件描述符在内核中指向同一个 `file` 结构体, 也就是说, `file` 结构体的引用计数要增加。

由于代码段(加载到内存的执行码)在内存中是只读的, 所以父、子进程可共用代码段, 而数据段和堆栈段子进程则完全从父进程复制了一份。

(2) 父进程进行 `fork` 系统调用时完成的操作

假设 `id=fork()`, 父进程进行 `fork` 系统调用时, `fork` 所做工作如下:

- ① 为新进程分配 `task_struct` 任务结构体内存空间。
- ② 把父进程 `task_struct` 任务结构体复制到子进程 `task_struct` 任务结构体。
- ③ 为新进程在其内存上建立内核堆栈。
- ④ 对子进程 `task_struct` 任务结构体中部分变量进行初始化设置。
- ⑤ 把父进程的有关信息复制给子进程, 建立共享关系。
- ⑥ 把子进程加入到可运行队列中。
- ⑦ 结束 `fork()` 函数, 返回子进程 ID 值给父进程中栈段变量 `id`。
- ⑧ 当子进程开始运行时, 操作系统返回 0 给子进程中栈段变量 `id`。

(3) `fork` 调用时所发生的事情

下面代码是 `fork` 函数调用模板, `fork` 函数调用后, 常与 `if-else` 语句结合使用, 使父、子进程执行不同的流程。假设下面的代码执行时产生的是 `x` 进程, `fork` 后产生的子进程是 `xx` 进程, `xx` 进程的进程 ID 号为 1000。

```
int pid ;
pid = fork();
if (pid < 0) {
    perror("fork failed");
    exit(1);
}
if (pid == 0) {
    message = "This is the child\n";
} else {
    message = "This is the parent\n";
}
```

调用 `fork` 前, 内存中只有 `x` 进程, 如图 12-9 所示, 此时 `xx` 进程还没“出生”。

调用 `fork` 后，内存中不仅有 `x` 进程（父进程），还有 `xx` 进程（子进程）。`fork` 的时候，系统几乎把父进程整个堆栈段（除代码段，代码段父、子进程是共享的）复制给了子进程，复制完成后，`x` 进程和 `xx` 进程是两个独立的进程，如图 12-10 所示，只不过 `x` 进程栈中变量 `id` 值此时为 1000，而 `xx` 进程栈中变量 `id` 值为 0。`fork` 调用完成后，`x` 进程由系统态回到用户态。此后，`x` 进程和 `xx` 进程各自都需要从自己代码段指针指向的代码点继续往下执行，父进程 `x` 往下执行时，判断 `id` 大于 0，所以执行大于 0 的程序段，而子进程 `xx` 往下执行时，判断 `id` 等于 0，所以执行等于 0 的程序段。

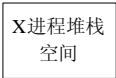


图 12-9 fork 前的内存

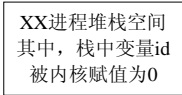
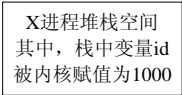


图 12-10 fork 后的内存

(4) fork 函数原型

fork 函数说明	
所需头文件	#include <sys/types.h> // 提供类型 pid_t 的定义 #include <unistd.h>
函数说明	建立一个新的进程
函数原型	pid_t fork(void)
函数返回值	0: 返回给子进程
	子进程的 ID (大于 0 的整数): 返回给父进程
	-1: 出错, 返回给父进程, 错误原因存于 errno 中
错误代码	EAGAIN: 内存不足
	ENOMEM: 内存不足, 无法配置核心所需的数据结构空间

(5) fork 函数使用实例

fork.c 源代码如下:

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    pid_t pid;
    char *message;
    int n;
    pid = fork();
    if (pid < 0) {
        perror("fork failed");
        exit(-1);
    }
    if (pid == 0) {
        message = "This is the child\n";
        n = 3;
    } else {
```

```
    wait(0) ; /*阻塞等待子进程返回*/
    message = "This is the parent\n";
    n = 1;
}
for( ; n > 0; n--) {
    printf(message);
    sleep(1);
}
return 0;
}
```

编译 `gcc fork.c -o fork`。

执行 `./fork`，执行结果如下：

```
This is the child
This is the child
This is the child
This is the parent
```

读者可以把 `sleep(1)` 改成 `sleep(30)`，然后通过 `ps -ef|grep fork` 查看进程数。

(6) fork 后程序处理的两种情形

一种为父进程希望复制自己，使父、子进程同时执行不同的代码段。这是网络并发服务端常见的模型，父进程等待客户端的服务请求，当这种请求到达时，父进程调用 `fork`，让子进程处理此请求，父进程则继续等待下一个服务请求。

另一种为 `fork` 后，通过 `exec` 执行另一个程序，在终端上执行命令属于这种情况，Shell 进程 `fork` 后，立即调用 `exec` 去运行执行命令。

(7) fork 之后处理文件描述符的两种常见情况

父进程等待子进程完成。在这种情况下，父进程无需对其文件描述符做任何处理，当子进程终止后，它曾进行过读/写操作的任一共享描述符的文件位移量已做了相应更新。

父、子进程各自执行不同的程序段。在这种情况下，在 `fork` 之后，父、子进程各自关闭它们不需要使用的文件描述符，并且不干扰对方使用的文件描述符。这种方式在并发网络服务器中经常使用到。

(8) 除了打开文件之外，很多父进程的其他性质也由子进程继承

- 实际用户 ID、实际组 ID、有效用户 ID、有效组 ID；
- 附加组 ID；
- 进程组 ID；
- 会话 ID；
- 控制终端；

- 设置-用户-ID 标志和设置-组-ID 标志;
- 当前工作目录;
- 根目录;
- 文件权限屏蔽字;
- 信号屏蔽和排列;
- 打开的文件描述符;
- 环境变量;
- 连接的共享存储段;
- 数据段、代码段、堆段、.bss 段;
- 资源限制。

(9) 父、子进程之间的区别

- fork 的返回值;
- 进程 ID;
- 不同的父进程 ID;
- 子进程的 tms_utime、tms_stime、tms_cutime 以及 tms_ustime 设置为 0;
- 父进程设置的锁, 子进程不继承;
- 未处理的闹钟信号子进程将清除;
- 子进程的未决告警被清除;
- 子进程的未决信号集设置为空集。

3. vfork 函数

由于 fork 完整地复制了父进程的整个堆栈空间, 因此执行速度是比较慢的。为了加快 fork 的执行速度, UNIX 系统设计者创建了 vfork, vfork 也能创建新进程, 但它不产生父进程的副本。vfork 用于创建一个新进程, 而该新进程的目的是 exec 一个新程序。

vfork 与 fork 一样都创建一个子进程, 但是它并不将父进程的堆栈空间完全复制到子进程中, 因为子进程会立即调用 exec (或 exit), 于是也就不会访问地址空间。不过在子进程调用 exec 或 exit 之前, 它在父进程的堆栈空间中运行。

vfork 保证子进程先运行, 在它调用 exec 或 exit 之后, 父进程才可能被调度运行, 如果在调用这两个函数之前, 子进程依赖于父进程的进一步动作, 则会导致死锁。

(1) vfork 函数原型

fork 函数说明	
所需头文件	<code>#include <sys/types.h></code> <code>#include <unistd.h></code>
函数说明	<code>vfork</code> 新建的子进程和父进程共享所有的资源，在子进程中对数据的修改也就是对父进程数据的修改，这里一定要注意。对于 <code>vfork</code> 所生成的父、子进程，父进程是在子进程调用了 <code>_exit()</code> 或者 <code>exe()</code> 后才会继续执行。不调用这两个函数则会出错
函数原型	<code>pid_t vfork(void)</code>
函数返回值	0: 返回给子进程
	子进程的 ID (大于 0 的整数): 返回给父进程
	-1: 出错, 返回给父进程, 失败原因记录在 <code>error</code> 中
错误代码	EAGAIN: 内存不足
	ENOMEM: 内存不足, 无法配置核心所需的数据结构空间

(2) vfork 函数举例

vfork.c 源代码如下:

```
#include <unistd.h>
#include <stdio.h>
int main()
{
    int i = 1 ;
    if ( vfork() == 0 )
    {
        printf("This is the child process\n") ;
        sleep(10) ;
        i = 2 ;
        printf("child process i = %d\n", i ) ;
        _exit(-1);
    } else {
        printf("This is the parent process\n") ;
        printf("parent process i = %d\n", i ) ;
    }
    return 0;
}
```

编译 `gcc vfork.c -o vfork`。

执行 `./vfork`, 执行结果如下:

```
This is the child process
child process i = 2
This is the parent process
parent process i = 2
```

(3) fork 与 vfork 的区别

使用 `fork` 调用, 会为子进程复制父进程所拥有的资源 (进程环境、栈堆等), 而 `vfork` 设计时要求子进程立即调用 `exec`, 而不修改任何内存, `vfork` 新建的子进程则是和父进程共享所有的资源, 在子进程中对数据的修改也就是对父进程数据的修改, 这一点一定要注意。

使用 `fork` 系统调用产生父子进程，在默认情况下无需人为干预，父子进程的执行顺序是由操作系统调度的，谁先执行并不确定。而对于 `vfork` 所生成的父子进程，父进程是在子进程调用了 `_exit` 或者 `exec` 后才会继续执行，不调用这两个函数，父进程会等待（父进程由于没有收到子进程表示已执行的相关信号所以进行等待）。

`vfork` 的出现是为了解决当初 `fork` 浪费用户内存空间的问题，因为在 `fork` 后，很有可能去执行 `exec` 函数重生，`vfork` 设计思想就是取消 `fork` 造成堆栈的复制，使用 `vfork` 可以避免资源的浪费，但是也带来了资源共享所产生的问题。

在 Linux 中，对 `fork` 进行了优化，调用时采用“写时复制”（COW, copy on write）的方式，在系统调用 `fork` 生成子进程的时候，不马上为子进程复制父进程的资源，而是在遇到“写入”（对资源进行修改）操作时才复制资源。它使得一个普通的 `fork` 调用非常类似于 `vfork`，但又避免了 `vfork` 的缺点，使得 `vfork` 变得没有必要。

4. exec 函数

(1) exec 函数说明

`fork` 函数用于创建一个子进程，该子进程几乎是父进程的副本，而有时，我们希望子进程去执行另外的程序，`exec` 函数族就提供了一个在进程中启动另一个程序执行的方法。它可以根据指定的文件名或目录名找到可执行文件，并用它来取代原调用进程的数据段、代码段和堆栈段，在执行完之后，原调用进程的内容除了进程号外，其他全部被新程序的内容替换了。另外，这里的可执行文件既可以是二进制文件，也可以是 Linux 下任何可执行的脚本文件。

(2) 在 Linux 中使用 exec 函数族的两种主要情况

- 当进程认为自己不能再为系统和用户做出任何贡献时，就可以调用任何 `exec` 函数族让自己重生。
- 如果一个进程想执行另一个程序，那么它就可以调用 `fork` 函数新建一个进程，然后调用任何一个 `exec` 函数使子进程重生。

(3) exec 函数族语法

实际上，在 Linux 中并没有 `exec` 函数，而是有 6 个以 `exec` 开头的函数族，下表列举了 `exec` 函数族的 6 个成员函数的语法。

exec（执行文件）	
所需头文件	#include <unistd.h>
函数说明	执行文件
函数原型	int execl(const char *path, const char *arg, ...)
	int execv(const char *path, char *const argv[])
	int execlp(const char *path, const char *arg, ..., char *const envp[])
	int execve(const char *path, char *const argv[], char *const envp[])
	int execlp(const char *file, const char *arg, ...)
	int execvp(const char *file, char *const argv[])

续表

exec（执行文件）	
函数返回值	成功：函数不会返回
	出错：返回-1，失败原因记录在 error 中

这 6 个函数在函数名和使用语法的规则上都有细微的区别，下面就从可执行文件查找方式、参数传递方式及环境变量这几个方面进行比较说明。

① 查找方式：上表其中前 4 个函数的查找方式都是完整的文件目录路径，而最后两个函数（也就是以 p 结尾的两个函数）只给出文件名，系统就会自动从环境变量“\$PATH”所指出的路径中进行查找。

② 参数传递方式：exec 函数族的参数传递有两种方式，一种是逐个列举的方式，而另一种则是将所有参数整体构造成指针数组进行传递。在这里参数传递方式是以函数名的第 5 位字母来区分的，字母为“l”（list）表示逐个列举的方式；字母为“v”（vector）表示将所有参数整体构造成指针数组传递，然后将该数组的首地址当做参数传给它，数组中的最后一个指针要求是 NULL。读者可以观察 execl、execle、execlp 与 execv、execve、execvp 语法的区别。

③ 环境变量：exec 函数族一般使用系统默认的环境变量，也可以传入指定的环境变量。这里以“e”（environment）结尾的两个函数 execle 和 execve 就可以在 envp[] 中指定环境变量，指定后的环境变量将替换掉该进程继承的所有环境变量。

（4）PATH 环境变量说明

PATH 环境变量包含了一张目录表，系统通过 PATH 环境变量定义的路径搜索执行码，PATH 环境变量定义时，目录之间需用“:”分隔，以“.”号表示结束。PATH 环境变量定义在用户的.profile 或.bash_profile 中，下面是 PATH 环境变量定义的样例，此 PATH 变量指定在“/bin”、“/usr/bin”和当前目录三个目录搜索执行码。

```
PATH=/bin:/usr/bin:.\nexport $PATH
```

（5）进程中的环境变量说明

在 Linux 中,Shell 进程是所有执行码的父进程。当一个执行码执行时,Shell 进程会 fork 子进程，然后调用 exec 函数去执行执行码。Shell 进程堆栈中存放着该用户下的所有环境变量，使用 execl、execv、execlp、execvp 函数使执行码重生时，Shell 进程会将所有环境变量复制给生成的新进程，而使用 execle、execve 时，新进程不继承任何 Shell 进程的环境变量，而由 envp[] 数组自行设置环境变量。

（6）exec 函数族关系

exec 函数名对应的含义		
第 4 位	统一为：exec	
第 5 位	1：参数传递为逐个列举方式	execl、execle、execlp

续表

exec 函数名对应的含义		
第 6 位	v: 参数传递为构造指针数组方式	execv、execve、execvp
	e: 可传递新进程环境变量	execle、execve
	p: 可执行文件查找方式为文件名	execlp、execvp

事实上，这 6 个函数中真正的系统调用只有 execve，其他 5 个都是库函数，它们最终都会调用 execve 这个系统调用，调用关系如图 12-11 所示。

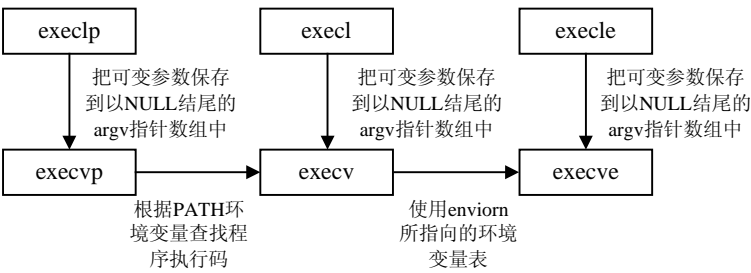


图 12-11 exec 函数族关系图

(6) exec调用举例

```
char *const ps_argv[]={ "ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL};
char *const ps_envp[] ={"PATH=/bin:/usr/bin", "TERM=console", NULL};
execl("/bin/ps", "ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL);
execvp("/bin/ps", ps_argv);
execle("/bin/ps", "ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL,
ps_envp);
execve("/bin/ps", ps_argv, ps_envp);
execlp("ps", "ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL);
execvp("ps", ps_argv);
```

请注意，exec 函数族形参展开时的前两个参数，第一个参数是带路径的执行码（execlp、execvp 函数第一个参数是无路径的，系统会根据 PATH 自动查找，然后合成带路径的执行码），第二个是不带路径的执行码，执行码可以是二进制执行码和 Shell 脚本。

(7) exec 函数族使用注意点

在使用 exec 函数族时，一定要加上错误判断语句。因为 exec 很容易执行失败，其中最常见的原因有以下三条：

- ① 找不到文件或路径，此时 errno 被设置为 ENOENT。
- ② 数组 argv 和 envp 记用 NULL 结束，此时 errno 被设置为 EFAULT。
- ③ 没有对应可执行文件的运行权限，此时 errno 被设置为 EACCES。

(8) exec 后新进程保持原进程特征

■ 环境变量（使用了 execle、execve 函数则不继承环境变量）；

- 进程 ID 和父进程 ID;
- 实际用户 ID 和实际组 ID;
- 附加组 ID;
- 进程组 ID;
- 会话 ID;
- 控制终端;
- 当前工作目录;
- 根目录;
- 文件权限屏蔽字;
- 文件锁;
- 进程信号屏蔽;
- 未决信号;
- 资源限制;
- tms_utime、tms_stime、tms_cutime 以及 tms_ustime 值。

对打开文件的处理与每个描述符的 `exec` 关闭标志值有关，进程中每个文件描述符有一个 `exec` 关闭标志 (`FD_CLOEXEC`)，若此标志设置，则在执行 `exec` 时，关闭该描述符，否则该描述符仍打开。除非特地用 `fcntl` 设置了该标志，否则系统的默认操作是在 `exec` 后仍保持这种描述符打开，利用这一点可以实现 I/O 重定向。

(9) `execlp` 函数举例

`execlp.c` 源代码如下：

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    if(fork()==0){
        if(execlp("/usr/bin/env", "env", NULL)<0)
        {
            perror("execlp error!");
            return -1 ;
        }
    }
    return 0 ;
}
```

编译 `gcc execlp.c -o execlp`。

执行 `./execlp`，执行结果如下：

```
HOME=/home/test
DB2DB=test
SHELL=/bin/bash
.....
```

由执行结果看出，`execlp` 函数使执行码重生时，继承了 Shell 进程的所有环境变量，其他三个不以 `e` 结尾的函数同理。

(10) `execle` 函数举例

利用函数 `execle`，将环境变量添加到新建的子进程中去。

`execle.c` 源代码如下：

```
#include <unistd.h>
#include <stdio.h>
int main()
{
    /*命令参数列表，必须以 NULL 结尾*/
    char *envp[]={"PATH=/tmp","USER=sun",NULL};
    if(fork()==0){
        /*调用 execle 函数，注意这里也要指出 env 的完整路径*/
        if(execle("/usr/bin/env","env",NULL,envp)<0)
        {
            perror("execle error!");
            return -1 ;
        }
    }
    return 0 ;
}
```

编译：`gcc execle.c -o execle`。

执行 `./execle`，执行结果如下：

```
PATH=/tmp
USER=sun
```

可见，使用 `execle` 和 `execve` 可以自己向执行进程传递环境变量，但不会继承 Shell 进程的环境变量，而其他 4 个 `exec` 函数则继承 Shell 进程的所有环境变量。

5. wait 和 waitpid 函数

(1) wait 函数说明

当一个进程正常或异常终止时，内核就向其父进程发送 `SIGCHLD` 信号。因为子进程终止是个异步事件，这种信号也是内核向父进程发的异步通知。父进程可以忽略该信号，或者提供一个该信号发生时即被调用执行的函数（信号处理程序）。

父进程同步等待子进程退出时，则调用 `wait` 函数，此时父进程可能会有如下三种情形：

- ① 阻塞（如果其所有子进程都还在运行）。
- ② 带回子进程的终止状态立即返回（如果已有一个子进程终止，正等待父进程取其终止状态）。
- ③ 出错立即返回（如果它没有任何子进程）。

(2) `wait` 与 `waitpid` 函数原型

wait（等待子进程的中断和结束）		
所需头文件	#include <sys/types.h> #include <sys/wait.h>	
函数说明	<code>wait()</code> 会暂时停止目前进程的执行，直到有信号来到或子进程结束。如果在调用 <code>wait()</code> 时子进程已经结束，则 <code>wait()</code> 会立即返回子进程结束状态值。子进程的结束状态值会由参数 <code>status</code> 返回，而子进程的进程识别码也会一起返回。如果不在意结束状态值，则 <code>status</code> 可以设成 <code>NULL</code>	
函数原型	pid_t wait (int *status)	
函数传入值	status	这里的 <code>status</code> 是一个整型指针，是该子进程退出时的状态： ■ <code>status</code> 若为空，则代表不记录子进程结束状态 ■ <code>status</code> 若不为空，则由 <code>status</code> 记录子进程的结束状态值 另外，子进程的结束状态可由 Linux 中一些特定的宏来测定
函数返回值	成功	返回子进程识别码（PID）
	出错	-1，失败原因存于 <code>errno</code> 中

waitpid（等待子进程的中断和结束）		
所需头文件	#include <sys/types.h> #include <sys/wait.h>	
函数说明	<code>waitpid()</code> 会暂时停止目前进程的执行，直到有信号来到或子进程结束。如果在调用 <code>waitpid()</code> 子进程已经结束，则 <code>waitpid()</code> 会立即返回子进程结束状态值。子进程的结束状态值会由参数 <code>status</code> 返回，而子进程的进程识别码也会一起返回。如果不在意结束状态值，则参数 <code>status</code> 可以设成 <code>NULL</code> 。参数 <code>pid</code> 为欲等待的子进程识别码	
函数原型	pid_t waitpid(pid_t pid,int * status,int options)	
函数传入值	pid	<-1: 等待进程组识别码为 <code>pid</code> 绝对值的任何子进程
		-1: 等待任何子进程，相当于 <code>wait()</code>
		0: 等待进程组识别码与目前进程相同的任何子进程
		>0: 等待任何子进程识别码为 <code>pid</code> 的子进程
	options	参数 <code>options</code> 可以为 0 或与下面的参数进行 OR 组合 WNOHANG: 如果没有任何已经结束的子进程则马上返回，不予以等待。此时返回值为 0 WUNTRACED: 如果子进程进入暂停执行情况则马上返回，但结束状态不予以理会
函数传出值	status	同 <code>wait</code> 函数
函数返回值	成功	返回子进程识别码（PID） 使用选项 <code>WNOHANG</code> 且没有子进程退出返回 0
	出错	-1，失败原因存于 <code>errno</code> 中

对 status 状态判断的宏	
说明	子进程的结束状态返回后存于 status
宏	宏意义说明
WIFEXITED(status)	如果子进程正常结束返回的值。取 exit 或 _exit 的低 8 位
WEXITSTATUS(status)	取得子进程 exit() 返回的结束代码，一般会先用 WIFEXITED 来判断是否正常结束才能使用此宏
WIFSIGNALED(status)	如果子进程是因为信号而结束则此宏值为真
WTERMSIG(status)	取得子进程因信号而中止的信号代码，一般会先用 WIFSIGNALED 来判断后才使用此宏
WIFSTOPPED(status)	如果子进程处于暂停执行情况，则此宏值为真。一般只有使用 WUNTRACED 时才会有此情况
WSTOPSIG(status)	取得引发子进程暂停的信号代码，一般会先用 WIFSTOPPED 来判断后才使用此宏

子进程的终止信息存放在一个 int 变量中，其中包含了多个字段位。用宏定义可以取出其中的每个字段位：如果子进程是正常终止的，WIFEXITED 取出的字段值非零，WEXITSTATUS 取出的字段值就是子进程的退出状态。如果子进程是收到信号而异常终止的，WIFSIGNALED 取出的字段值非零，WTERMSIG 取出的字段值就是信号的编号。

(3) wait 和 waitpid 两函数的说明

如果父进程的所有子进程都还在运行，调用 wait 将使父进程阻塞，而调用 waitpid 时，如果在 options 参数中指定 WNOHANG，可以使父进程不阻塞而立即返回 0。

wait 等待第一个终止的子进程，而 waitpid 可以通过 pid 参数指定等待哪一个子进程。

当 pid=-1、option=0 时，waitpid 函数等同于 wait，可以把 wait 看做 waitpid 实现的特例。

可见，调用 wait 和 waitpid 不仅可以获得子进程的终止信息，还可以使父进程阻塞等待子进程终止，起到进程间同步的作用。如果参数 status 不是空指针，则子进程的终止信息通过这个参数传出，如果只是为了同步而不关心子进程的终止信息，可以将 status 参数指定为 NULL。

waitpid 函数提供了 wait 函数没有提供的三个功能：

- ① waitpid 等待一个特定的进程，而 wait 则返回任一终止子进程的状态。
- ② waitpid 提供了一个 wait 的非阻塞版本，有时希望取得一个子进程的状态，但不想进程阻塞。
- ③ waitpid 支持作业控制。

(4) wait 函数使用实例

wait.c 源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
```

```

#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main()
{
    pid_t pid;
    int status,i;
    if(fork()==0){
        printf("This is the child process pid =%d\n",getpid());
        exit(5);
    }else{
        sleep(1);
        printf("This is the parent process,wait for child...\n");
        pid=wait(&status);
        i=WEXITSTATUS(status);
        printf("child pid =%d, exit status=%d\n",pid,i);
    }
    return 0 ;
}

```

编译 gcc wait.c -o wait。

执行 ./wait, 执行结果如下:

```

This is the child process pid =6904
This is the parent process ,wait for child...
child pid =6904, exit status=5

```

(5) waitpid 函数使用实例

waitpid.c 源代码如下:

```

#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    pid_t pid;
    pid = fork();
    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }
    if (pid == 0) { //子进程
        int i;
        for (i = 3; i > 0; i--) {
            printf("This is the child\n");
            sleep(1);
        }
        exit(3);
    } else { //父进程
        int stat_val;

```



```
waitpid(pid, &stat_val, 0); /*阻塞等待子进程*/
if (WIFEXITED(stat_val))
    printf("Child exited with code %d\n", WEXITSTATUS(stat_val));
else if (WIFSIGNALED(stat_val))
    printf("Child terminated abnormally, signal %d\n", WTERMSIG(stat_val));
}
return 0;
}
```

编译 gcc waitpid.c -o waitpid。

执行 ./waitpid，执行结果如下：

```
This is the child
This is the child
This is the child
Child exited with code 3
```

6. 进程中的三种用户 ID

(1) 进程中三种用户 ID 的含义

表 12-2 列出了进程执行时，与进程相关联的三种用户 ID，这三种 ID 在 Linux 书中经常提及，但同时也是易混 不好理解的地方。

表 12-2 三种用户 ID 含义表

与每个进程相关联的用户 ID 和组 ID	
实际用户 ID	我们实际上是谁，ID 号保存的是启动进程用户 ID 和组 ID
实际组 ID	
有效用户 ID	用于文件存取许可权检查。当执行码设置了设置-用户-ID（set-user-ID）位时，此时进程的有效用户为该文件所属用户，同时就获取了所属用户的用户权限。有效组同理
有效组 ID	
保存的设置-用户-ID	用来保存有效用户 ID 和有效组 ID 的副本
保存的设置-组-ID	

实际用户 ID 和实际组 ID 标识我们究 是谁，这两个字段是用户登录时取自口令文件中的登录项。通常，在一个登录会话期间，这些值并不改变，但是超级用户进程可以对这两个 ID 值随便改。

有效用户 ID、有效组 ID 决定了文件访问权限。有效用户 ID、有效组 ID 主要在校验文件权限时使用，比如打开文件、创建文件、修改文件、kill 别的进程等。

保存的设置-用户-ID 和设置-组-ID 在执行一个程序时，保存了有效用户 ID 和有效组 ID 的副本。

通常，当执行一个程序文件时，进程的有效用户 ID 通常就是实际用户 ID，有效组 ID 通常是实际组 ID。

每个文件有一个所有者（属主）和组所有者（属组），所有者由 stat 结构（文件属性结构）中的 st_uid 表示，组所有者则由 st_gid 成员表示。当在文件方式字（st_mode）中设置一个

特殊标志时(其含义是“当执行此文件时,将进程的有效用户 ID 设置为文件的所有者(st_uid)”),有效用户 ID 就不一定等于实际用户 ID。与此相类似,在文件方式字中可以设置另一位,它使得执行此文件进程的有效组 ID 设置为文件的组所有者(st_gid)。在文件方式字中的这两位被称之为设置-用户-ID(set-user-ID)位和设置-组-ID(set-group-ID)位。注意设置-用户-ID 位和保存的设置-用户-ID 是两个不同的字段,前一个是一个特殊标志,后一个用来保存有效用户 ID 的副本。

例如,假设 Y 用户有一执行码 test,此时 test 的设置-用户-ID(set-user-ID)位没有设置。X 用户执行 Y 用户的执行码 test 时,此时 test 实际用户 ID 等于 X 用户 ID,有效用户 ID 等于 X 用户 ID,保存的设置-用户-ID 等于 X 用户 ID,三类组 ID 同理。由于有效用户 ID 决定了执行码的用户权限,所以 test 执行码是 X 用户权限,有权限读取和修改 X 用户的文件,而对 Y 用户的文件则不一定有权限。

当 test 的设置-用户-ID(set-user-ID)位进行了设置时,X 用户执行 Y 用户的执行码 test,此时 test 实际用户 ID 等于 X 用户 ID,有效用户 ID 等于 Y 用户 ID,保存的设置-用户-ID 等于 Y 用户 ID,三类组 ID 同理。此时 test 执行码为 Y 用户权限,有权限读取和修改 Y 用户的文件,而对 X 用户的文件不一定有权限。如果要让 test 某一段时间内有 X 用户的权限,则可用 seteuid(getuid())方法把有效用户 ID 修改为实际用户 ID(即 X 用户 ID),随后 test 又想恢复 Y 用户权限,则可把有效用户 ID 重置,因为设置-用户-ID 是以前有效用户 ID 的副本,在非特权用户下,系统会根据设置-用户-ID 的值判断此有效用户 ID 设置是否允许。在非特权用户下,有效用户 ID 重新设置时,只能等于实际用户 ID 或保存的设置-用户-ID。

(2) setuid 函数和 setgid 函数原型

setuid (设置真实的用户识别码)		
所需头文件	#include <unistd.h>	
函数说明	setuid()用来重新设置执行目前进程的用户识别码。不过,要让此函数有作用,其有效的用户识别码必须为 0 (root)。在 Linux 下,当 root 使用 setuid()来转换成其他用户识别码时,root 权限会被抛弃,完全转换成该用户身份,也就是说,该进程往后将不再具有可 setuid()的权利。如果只是想暂时抛弃 root 权限, 后想重新取回权限,则必须使用 seteuid()	
函数原型	int setuid(uid_t uid)	
函数传入值	uid	设置实际用户 ID 号
	成功	0
函数返回值	出错	-1, 失败原因存于 errno 中
	一般在编写具有 setuid root 的程序时,为减少此类程序带来的系统安全风险,在使用完 root 权限后,建议马上执行 setuid(getuid());来抛弃 root 权限	

setgid (设置真实的组织识别码)		
所需头文件	#include <unistd.h>	
函数说明	setgid()用来将目前进程的真实组织识别码 (real gid) 设成参数 gid 值。如果是超级用户身份执行此调用,则 real、effective 与 saved gid 都会设成参数 gid	

续表

setgid (设置真实的组织识别码)		
函数原型	int setgid(gid_t gid)	
函数传入值	gid	设置实际组 ID 号
函数返回值	成功	0
	出错	-1, 失败原因存于 errno 中
错误代码	EPERM: 并非以超级用户身份调用, 而且参数 gid 并非进程的 effective gid 或 saved gid 值之一	

(3) setuid 和 setgid 函数使用说明

可以用 setuid 函数设置实际用户 ID 和有效用户 ID。与此类似, 可以用 setgid 函数设置实际组 ID 和有效组 ID。

使用 setuid 改变用户 ID 的规则:

- ① 若进程具有超级用户特权, 则 setuid(uid) 执行时将实际用户 ID、有效用户 ID, 以及保存的设置-用户-ID 设置为 uid。
- ② 若进程没有超级用户特权, 但是 uid 等于实际用户 ID 或保存的设置-用户-ID, 则 setuid(uid) 执行时只将有效用户 ID 设置为 uid, 不改变实际用户 ID 和保存的设置-用户-ID。
- ③ 如果上面两个条件都不满足, 则 errno 设置为 EPERM, 并返回出错。

关于内核所维护的三个用户 ID 的说明:

- ① 只有超级用户进程可以更改实际用户 ID。通常, 实际用户 ID 是在用户登录时, 由 login 程序设置的, 而且决不会改变它。因为 login 是一个超级用户进程, 当它调用 setuid 时, 设置所有三个用户 ID。
- ② 仅当对程序文件设置了设置-用户-ID 位时, exec 函数才会设置有效用户 ID。如果设置-用户-ID 位没有设置, 则 exec 函数不会改变有效用户 ID, 而会将其维持在原先值。任何时候都可以调用 setuid, 将有效用户 ID 设置为实际用户 ID 或保存的设置-用户-ID。
- ③ “保存的设置-用户-ID” 是进程 exec 时从有效用户 ID 复制而来, 复制后将此副本保存起来。

表 12-3 列出了改变三种用户 ID 的不同方法表。从表中可以看出, 有效用户 ID 是主角, 因为有效用户 ID 决定了进程对文件的访问权限, 非特权用户下, 有效用户 ID 的值重新设置时, 只能等于实际用户 ID 或保存的设置-用户-ID。

表 12-3 改变三种用户 ID 的不同方法表

ID	exec		setuid(uid)	
	设置-用户-ID 位关闭	设置-用户-ID 位打开	超级用户	非特权用户
实际用户 ID	不变	不变	设为 uid	不变
有效用户 ID	不变	设置为程序文件的用户 ID	设为 uid	设为 uid
保存的设置-用户-ID	从有效用户 ID 复制	从有效用户 ID 复制	设为 uid	不变

(4) seteuid 和 setegid 函数原型

seteuid (设置有效的用户识别码)		
所需头文件	#include <unistd.h>	
函数说明	seteuid()用来重新设置执行目前进程的有效用户识别码。在 Linux 下, seteuid(euid)相当于 setreuid(-1,euid)	
函数原型	int seteuid(uid_t euid)	
函数传入值	euid	设置有效用户的 ID 号
	成功	0
函数返回值	出错	-1, 失败原因存于 errno 中

setegid (设置有效的组织识别码)		
所需头文件	#include <unistd.h>	
函数说明	设置执行目前进程的有效组识别号	
函数原型	int setegid(gid_t egid)	
函数传入值	egid	设置有效组 ID 号
	成功	0
函数返回值	出错	-1, 失败原因存于 errno 中

(5) seteuid 和 setegid 函数说明

seteuid 和 setegid 只更改有效用户 ID 和有效组 ID。

执行 seteuid(uid)时, 对于一个非超级用户, 只有 uid 等于实际用户 ID 或其保存的设置-用户-ID 时才能设置。对于一个超级用户, 则可将有效用户 ID 设置为 uid (这区别于 setuid 函数, 特权用户下它更改的是三个用户 ID)。

(6) 函数举例说明几种 ID 的作用

在 x 用户下编写 setid.c 源文件, 内容如下。

```
#include <unistd.h>
#include <pwd.h>
#include <sys/types.h>
#include <stdio.h>
int main(int argc,char **argv)
{
    pid_t my_pid,parent_pid;
    uid_t my_uid,my_euid;
    gid_t my_gid,my_egid;
    my_uid=getuid();
    my_euid=geteuid();
    my_gid=getgid();
    my_egid=getegid();
    printf("User ID%d\n",my_uid);
    printf("Effective User ID%d\n",my_euid);
```

```

printf("Group ID%ld\n",my_gid);
printf("Effective Group ID%ld\n",my_egid) ;
system("wc -l xx") ;

seteuid(getuid()) ;
setegid(getgid()) ;
my_uid=getuid();
my_euid=geteuid();
my_gid=getgid();
my_egid=getegid();

printf("after seteuid(getuid())\n");
printf("User ID%ld\n",my_uid);
printf("Effective User ID%ld\n",my_euid);
printf("Group ID%ld\n",my_gid);
printf("Effective Group ID%ld\n",my_egid) ;
system("wc -l xx") ;

return 0 ;
}

```

① 编译 `gcc setid.c -o setid`。

② 设置执行码设置-用户-ID (`set-user-ID`) 位和设置-组-ID (`set-group-ID`) 位。

```

$chmod u+s setid
$chmod g+s setid

```

③ 用 `vim xx` 文件输入几行内容，用 `chmod 700` 设置 `xx` 文件的权限。

④ 用 `id` 命令查看 `x` 用户的用户 ID 和组 ID，结果如下：

```
uid=1008(X) gid=1003(XX)
```

⑤ 到 `Y` 用户下，用 `id` 命令查看 `Y` 用户的用户 ID 和组 ID，结果如下：

```
uid=1016(Y) gid=1001(YY)
```

⑥ 在 `Y` 用户到相应目录执行 `./setid`，执行结果如下：

```

User ID1016
Effective User ID1008
Group ID1001
Effective Group ID1003
3 xx
after seteuid(getuid())
User ID1016
Effective User ID1016
Group ID1001
Effective Group ID1001
wc: xx: Permission denied

```

7. system 函数

(1) system 实现说明

system 在其实现中调用了 fork、exec 和 waitpid，因此有三种返回值：

- ① 如果 fork 失败或者 waitpid 返回除 EINTR 之外的出错，则 system 返回-1，而且 errno 中设置了错误类型。
- ② 如果 exec 失败，则其返回值如同 Shell 执行了 exit(127)一样。
- ③ 如果三个函数 (fork、exec 和 waitpid) 都执行成功，system 的返回值是执行 Shell 命令的终止状态，其状态值同 waitpid 中的说明。

使用 system，而不是直接使用 fork 和 exec 的优点是使用方便，而且 system 进行了所需的各种出错处理，以及各种信号处理。

(2) system 函数原型

system（执行 Shell 命令）	
所需头文件	#include <stdlib.h>
函数说明	system()会调用 fork()产生子进程，由子进程来调用/bin/sh-c string 来执行参数 string 字符串所代表的命令，此命令执行完后，随即返回原调用的进程。在调用 system()期间，SIGCHLD 信号会被暂时搁置，SIGINT 和 SIGQUIT 信号则会被忽略
函数原型	int system(const char * string)
函数返回值	如果 system()在调用/bin/sh 时失败，则返回 127，其他失败原因返回-1。若参数 string 为空指针(NULL)，则返回非零值。如果 system()调用成功则最后会返回执行 Shell 命令后的返回值，但是此返回值也有可能为 system()调用/bin/sh 失败所返回的 127，因此最好能再检查 errno 来确认是否执行成功
附加说明	在编写具有 SUID/SGID 权限的程序时，请 使用 system()，system()会继承环境变量，通过环境变量可能会造成系统安全的问题

(3) system 函数使用实例

system.c 源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    system("ls -al /etc/passwd /etc/shadow") ;
    return 0 ;
}
```

编译 gcc system.c -o system。

执行 ./system，执行结果如下：

```
-rw-r--r-- 1 root root 2676 2005-12-08 05:48 /etc/passwd
-rw-r----- 1 root shadow 2268 2005-12-08 05:48 /etc/shadow
```

12.4 进程关系

1. 终端

在 Linux 系统中，用户通过终端登录系统后得到一个 Shell 进程，这个终端成为 Shell 进程的控制终端（Controlling Terminal），Shell 进程启动的其他进程的控制终端也是这个终端。默认情况下（没有重定向），每个进程的标准输入、标准输出和标准错误输出都指向控制终端，进程从标准输入读也就是读用户的键盘输入，进程往标准输出或标准错误输出写也就是输出到显示器上。此外，在控制终端输入一些特殊的控制键可以给前台进程发信号，如 Ctrl+C 表示 SIGINT，Ctrl+\ 表示 SIGQUIT。

2. 会话与进程组

每次用户登录终端时，会产生一个会话（session）。从用户登录开始到用户退出为止，这段时间内在该终端执行的进程都属于这一个会话。

每个进程除了有一进程 ID 之外，还属于一个进程组（Process Group）。进程组是一个或多个进程的集合，每个进程组有一个唯一的进程组 ID。多个进程属于进程组的情况是多个进程用管道“|”号连接进行执行。如果在命令行执行单个进程时，这个进程组只有这一个进程。

3. 控制终端

在终端（包括 telnet 等伪终端）登录就会产生一个会话，此会话拥有这一个单独的控制终端。

建立与控制终端连接的会话首进程，被称之为控制进程，也就是 Shell 进程。

一个会话中的几个进程组可被分成一个前台进程组以及一个或几个后台进程组。

4. 作业控制

前后台运行的进程组又称为作业（Job），一个前台作业可以由多个进程组成，一个后台作业也可以由多个进程组成，Shell 进程可以同时运行一个前台作业和任意多个后台作业，这称为作业控制（Job Control）。

作业控制允许在一个终端上起多个作业（进程组），控制哪一个作业可以存取该终端，以及哪些作业在后台运行。

从 Shell 进程使用作业控制功能角度观察，可以在前台启动一个作业或后台启动多个作业。一个作业只是几个进程的集合，通常用管道连接各进程。

例如，下面的命令在后台（&表示在后台运行）启动了两个作业，这两个后台作业所调用的进程都在后台运行。

```
cat *.c | pg &
```

```
make all &
```

实际上有三个特殊字符可使终端驱动程序产生信号，信号将送至前台进程组的所有进程，而后台进程组作业则不受影响，它们是：

- ① 中断字符（DELETE 或 Ctrl+C）产生 SIGINT 信号。
- ② 退出字符（Ctrl+\）产生 SIGQUIT 信号。
- ③ 挂起字符（Ctrl+Z）产生 SIGTSTP 信号。

5. 会话与进程组的关系

用以下命令启动 5 个进程。

```
$ proc1 | proc2 &  
$ proc3 | proc4 | proc5
```

其中，proc1 和 proc2 属于同一个后台进程组，proc3、proc4、proc5 属于同一个前台进程组，Shell 进程本身属于一个单独的进程组。这些进程组的控制终端相同，它们属于同一个会话。其进程、进程组、会话的关系如图 12-12 所示。

现在从会话和进程组的角度重新来看登录和执行命令的过程。在上面的例子中，proc3、proc4、proc5 被 Shell 放到同一个前台进程组，其中，proc3 进程是该进程组的组长，Shell 进程调用 wait 等待它们运行结束，一旦它们全部运行结束，Shell 进程就调用 tcsetpgrp 函数将自己提到前台继续接收命令。但是注意，如果 proc3、proc4、proc5 中的某个进程又 fork 出子进程，子进程也属于同一进程组，但是 Shell 进程并不知道子进程的存在，也不会调用 wait 等待它结束。换句话说，proc3 | proc4 | proc5 是 Shell 进程的作业，而这个子进程不是，这是作业和进程组在概念上的区别。一旦前台作业运行结束，Shell 进程就把自己提到前台。

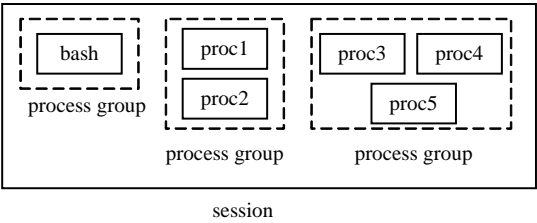


图 12-12 会话与进程组关系

6. 进程组和会话实例说明

(1) 进程组与会话实例一

```
$ ps -o pid,ppid,pgrp,session,tpgid,comm | cat  
PID PPID PGRP SESS TPGID COMMAND  
6994 6989 6994 6994 8762 bash  
8762 6994 8762 6994 8762 ps  
8763 6994 8762 6994 8762 cat
```

这个作业由 ps 和 cat 两个进程组成，在前台运行。从 PPID 列可以看出，这两个进程的父进程是 bash；从 PGRP 列可以看出，bash 在进程组 ID 为 6994 的进程组中，这个进程组 ID 等于 bash 的进程 ID，所以它是进程组的组长，而两个子进程在 8762 的进程组中，ps 是这个进程

组的组长；从 `SESS` 列可以看出这三个进程都在同一会话中，会话 ID 为 6994，`bash` 是会话首进程；从 `TPGID` 列可以看出，前台进程组 ID 是 8762，也就是 `ps` 和 `cat` 这两个进程所在的进程组，其中 `ps` 进程为进程组的组长。

(2) 进程组与会话实例二

```
$ ps -o pid,ppid,pgrp,session,tpgid,comm | cat &
[1] 8835
$  PID PPID PGRP  SESS TPGID COMMAND
   6994  6989  6994   6994   6994  bash
   8834  6994  8834   6994   6994  ps
   8835  6994  8834   6994   6994  cat
```

这个作业由 `ps` 和 `cat` 两个进程组成，在后台运行。`bash` 不等作业结束就打印提示信息“[1] 8835”，然后给出提示符接受新的命令，“1”是作业的编号，如果同时运行多个作业，可以用这个编号区分，“8835”是该作业中某个进程的进程 ID。

7. 进程组函数

(1) 进程组函数说明

每个进程组有一个组长进程，组长进程 ID 等于其进程组 ID。只要在某个进程组中有一个进程存在，则该进程组就存在，这与其组长进程是否终止无关。从进程组创建开始到其中的最后一个进程离开为止的时间区间称为进程组的生命期。

(2) 取进程组 ID 函数原型

getpgrp（取得进程组识别码）	
所需头文件	#include <unistd.h>
函数说明	getpgrp()用来取得目前进程所属的组织识别码，此函数相当于调用 getpgid(0)
函数原型	pid_t getpgrp(void)
函数返回值	返回目前进程所属的组织识别码

(3) 设置进程组 ID 函数原型

setpgid（设置进程组识别码）	
所需头文件	#include <unistd.h>
函数说明	setpgid()将参数 pid 指定进程所属的组织识别码设为参数 pgid 指定的组织识别码。如果参数 pid 为 0，则会用来设置目前进程的组识别码，如果参数 pgid 为 0，则会以目前进程的进程识别码来取代
函数原型	int setpgid(pid_t pid,pid_t pgid)
函数返回值	执行成功则返回组识别码，如果有错误则返回-1，错误原因存于 errno 中
错误代码	EINVAL: 参数 pgid 小于 0 EPERM: 进程权限不足，无法完成调用 ESRCH: 找不到符合参数 pid 指定的进程

进程调用 `setpgid(pid, pgid)`可以参加一个现存的组或者创建一个新进程组，这时 `pid` 进程的进程组 ID 设置为 `pgid`。如果这两个参数相等，则 `pid` 指定的进程变成进程组组长。

一个进程只能为它自己或它的子进程设置进程组 ID。在它的子进程调用了 `exec` 后，它就不再能改变该子进程的进程组 ID。如果设置的 `pid` 等于 0，则使用调用者的进程 ID；如果设置 `pgid` 等于 0，则使用 `pid` 进程的进程组 ID。

在大多数作业控制中，在 `fork` 之后调用此函数，使父进程设置其子进程的进程组 ID，然后使子进程设置其自己的进程组 ID。这些调用中有一个是 余的，但这样做可以保证父、子进程在进一步操作之前，子进程都进入了该进程组。如果不这样做的话，那么就产生一个竞态条件，因为它依赖于哪一个进程先执行。

在发送一个信号时，信号可以发送给一个进程或送给一个进程组。

8. 会话函数

(1) 会话函数原型

setsid（创建一个新的会话）	
所需头文件	<code>#include <sys/types.h></code> <code>#include <unistd.h></code>
函数说明	创建一个新的会话
函数原型	<code>pid_t setsid(void)</code>
函数返回值	若成功则为进程组 ID，若出错则为-1

(2) 会话说明

用户每次进行用户登录会产生一个会话。如果调用 `setsid`，此函数的进程不是一个进程组的组长，则此函数会创建一个新会话，所产生的结果如下：

- ① 此进程变成该新会话的会话首进程（会话首进程是创建该会话的进程），此进程是该新会话中的唯一进程。
- ② 此进程成为一个新进程组的组长进程，新进程组 ID 是此进程的进程 ID。
- ③ 此进程没有控制终端，如果在调用 `setsid` 之前，此进程有一个控制终端，那么这种联系也被解除。
- ④ 如果此调用进程已经是一个进程组的组长，则此函数返回出错。为了保证不处于这种情况下，通常先调用 `fork`，然后使其父进程终止，而子进程则继续。因为子进程继承了父进程的进程组 ID，而其进程 ID 则是新分配的，两者不可能相等，所以这就保证了子进程不是一个进程组的组长。

(3) setsid 函数的作用

- ① 让进程 脱原会话的控制。
- ② 让进程 脱原进程组的控制。

③ 让进程 脱离控制终端的控制。

9. 孤儿进程

一个父进程已终止的进程称为 孤儿进程 (orphan process)，这种进程由 1 号进程 init 收养。

10. 僵尸进程

(1) 僵尸进程的产生

一个进程在调用 `exit` 函数结束自己生命的时候，其实它并没有真正地被完全销毁，而是留下一个称为僵尸进程 (Zombie) 的数据结构。

在 Linux 进程的状态中，僵尸进程是非常特殊的一种，它已经释放了几乎所有内存空间，没有任何可执行代码，也不能被调度，仅仅在进程列表中保留一个表项，记载该进程的退出状态等信息供其他进程收集。除此之外，僵尸进程不再占有任何内存空间。它需要它的父进程来为它“收尸”，如果它的父进程没有设置 `SIGCHLD` 信号处理函数，或者没有设置 `SIGCHLD` 信号为忽略 (`SIG_IGN`)，又或者没有调用 `wait` (或 `waitpid`) 等待子进程结束，那么它就一直保持僵尸状态。如果这时父进程结束了，那么 `init` 进程会自动接手这个子进程，僵尸进程消失。但是如果父进程是一个循环，不会结束，那么子进程就会一直保持僵尸状态，这就是为什么系统中有时会有很多的僵尸进程的原因。

(2) 怎样来清除僵尸进程

清除僵尸进程有三种方法，具体说明如下：

① 改写父进程，在子进程死后为它“收尸”。具体做法是接管 `SIGCHLD` 信号，子进程死后，会发送 `SIGCHLD` 信号给父进程，父进程调用 `wait` (或 `waitpid`) 函数为子进程收尸。

② 在父进程中设置 `SIGCHLD` 信号处理函数或者设置 `SIGCHLD` 信号为忽略 (`SIG_IGN`)。

③ 把父进程杀掉，父进程死后，僵尸进程成为“孤儿进程”，过继给 1 号进程 `init`，`init` 始终会负责清理僵尸进程，它产生的所有僵尸进程也跟着消失。

12.5 守护进程与多进程并发案例

12.5.1 守护进程的编写

守护进程，也就是通常所说的 `Daemon` 进程 (又称精灵进程)，是 Linux 中的后台服务进程。它是一个生存期较长的进程，通常独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。Linux 系统有很多守护进程，大多数服务都是通过守护进程实现的，如作业规划进程 `crond`、打印进程 `lpd` 等。

在 Linux 终端执行进程时，当终端被关闭时，相应的进程都会自动关闭。但是守护进程却能

够突破这种限制，它从被执行开始运转，直到整个系统关闭时才会退出。如果想让某个进程不因为用户、终端、或其他的变化而受到影响，那么就必须把这个进程变成一个守护进程。

（1）守护进程的编写步骤

① 创建子进程，父进程退出。

首先调用 `fork`，然后使父进程调用 `exit` 函数退出，这一步给 Shell 进程造成程序运行完成的假象，让 Shell 进程把自己提到会话前台，此时用户在 Shell 终端里可以执行其他命令，从而让程序形式上脱离了控制终端。

由于父进程已经先于子进程退出，会造成子进程没有父进程，此时子进程变成一个孤儿进程。在 Linux 中，每当系统发现一个孤儿进程，就会自动由 1 号进程（init 进程）收养它，这样，子进程就会成为 init 进程的子进程。

② 调用 `setsid` 创建一个新会话。

在上一步调用 `fork` 函数后，子进程几乎是父进程的副本，虽然父进程退出了，但会话 ID、进程组 ID 等并没有改变，因此，还不是真正意义上脱离控制终端，而 `setsid` 函数能够使进程完全独立出来。

调用 `setsid` 函数后，该子进程成为新会话的首进程，成为一个新进程组的首进程而且没有控制终端。

③ 再次 `fork` 确保子进程不是会话首进程

此进程已经成为无终端的会话组长，但它可以重新申请打开一个控制终端。可以通过使进程不再是会话组长来禁止进程重新打开控制终端，所以再次 `fork` 使父进程退出，子进程确保不是会话首进程，子进程将永远不会重获控制终端。

④ 调用 `chdir` 函数改变当前工作目录

使用 `fork` 创建的子进程继承了父进程的当前工作目录。调用函数 `chdir("/")` 将当前工作目录更改为根目录，这样进程不使用任何目录，否则守护进程可能一直占用某个目录，这可能会造成超级用户不能卸载一个文件系统的情况。

⑤ 重设文件权限掩码

由于使用 `fork` 系统调用，新建的子进程继承了父进程的文件权限掩码，这就给该子进程使用文件带来了诸多的麻烦。因此，把文件权限掩码设置为 0，可以大大增强该守护进程的灵活性，设置方法为 `umask(0)`。

⑥ 关闭文件描述符

由于子进程从其父进程继承来的某些文件描述符通常需要关闭，究竟关闭哪些描述符则与具体的守护进程有关，文件描述符 0、1 和 2 所指的这 3 个文件已经失去了存在的价值，一般应被关闭。

⑦ 重定向 0、1、2 三个文件描述符

出于安全以及健壮性考虑,即使当前进程不使用 `stdin`、`stdout`、`stderr`,也应重新打开 0、1、2 三个文件描述符,使之指向 `/dev/null`。当然,也可以根据需要使之对应不同的(伪)文件。总之,最好保持 0、1、2 三个文件描述符呈现打开状态,并使之指向无 文件。

⑧ 处理 SIGCHLD 信号

处理 SIGCHLD 信号并不是必需的。但对于某些进程(如并发服务器进程),客户端在请求到来时, `fork` 子进程来处理请求。如果父进程不等待子进程结束,那么子进程将成为僵尸进程,而父进程等待子进程结束,将增加父进程的负担,影响服务器进程的并发性能。这时可以简单将 SIGCHLD 信号操作设为 SIG_IGN(表示忽略信号)来避免僵尸进程。设置方法如下:

```
signal(SIGCHLD,SIG_IGN);
```

(2) 守护进程代码举例

dameon.c 源代码如下:

```
/*创建守护进程实例*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
/*进程打开最多文件数,根据内核参数有关*/
#define MAXFILE 512
int main()
{
    pid_t pc;
    int i,fd,len;
    int j ;
    /*第一步,fork 进程,父进程退出*/
    pc=fork();
    if(pc<0){
        printf("error fork\n");
        exit(1);
    }else if(pc>0){
        exit(0);
    }
    /*第二步,设置新的会话进程,脱原终端会话的控制*/
    setsid();
    /*第三步,再次 fork,使之成为无会话的进程*/
    pc=fork();
    if(pc<0){
        printf("error fork\n");
        exit(1);
    }else if(pc>0){
        exit(0);
    }
    /*第四步,设置工作路径*/
```

```

chdir("/");
/*第五步,设置文件掩码*/
umask(0);
/*第六步,关闭继承的打开文件描述符*/
for(i=0;i<MAXFILE;i++)
    close(i);
/*第七步,重定向 0、1、2 文件描述符到零设备*/
j = open("/dev/null", O_RDWR );
dup2( j, 0 );
dup2( j, 1 );
dup2( j, 2 );
/*第八步,忽略 SIGCHLD 信号*/
signal(SIGCHLD,SIG_IGN);
/*这时创建完守护进程,以下开始正式进入守护进程工作*/
while(1){
    /*处理内容*/
    sleep(3) ;
}
return 0 ;
}

```

编译 `gcc dameon.c -o dameon`。

执行 `./dameon`, 用 `ps -ef | grep dameon` 查看守护进程的运行, 运行结果如下:

```
zjkf      7967      1 88 15:38 ?          00:00:09 ./dameon
```

12.5.2 多进程并发项目案例

(1) 项目说明

该项目是笔者参与的某省建设银行省分行数据仓库项目, 本节案例主要说明该项目的月末程序在多进程并发上的实现。下面是该案例的简要说明, 以及对该案例在硬件、数据库配置、数据库建库脚本、程序上调优的简要介绍。

月末程序主要功能是计算各个客户的存、贷、消费、贡献度, 然后按照指标把客户分成一系列的等级(如金卡、白金、银卡客户等), 最后把优质客户分配给各个客户经理进行个性化营销。

该项目月末模块计算量很大, 但需要月末某天 10 小时内完成计算, 所以需要利用软硬件上的各种性能。

在硬件上, 使用多 CPU、高内存, 磁盘阵列采用 0+1 方式, 将表空间分布到不同的磁盘阵列上。在数据库管理配置上, 使用非日志方式, 同时把数据缓冲区配置相当大(总内存的 $1/3 \sim 1/2$)。在建表脚本上, 将大表建立在多个表空间上, 发挥磁盘阵列的并发功能, 同时数据块的配置应尽可能大, 使用大数据块能让磁盘阵列一次能顺序读出大量的数据页。建表脚本使用页锁而不使用行级锁。

对于大数据量的处理, 在程序上处理优化也显得尤为重要, 下面是程序优化的说明。

该行客户信息表和存 表都有几千万级的数据量，所以单进程计算效率上是不允许的，只能采用多进程并发。程序设计时取了最大客户号，按照定制进程数，将最大客户号/进程数划分客户号段分配给各进程，然后每个进程按照客户号范围并发将大表数据导入到十几个小表内，导入完后对小表建索引，然后利用十几个进程进行并发计算。

由于月末处理程序划分为 20 多个子交易顺序完成，程序使用一个控制表来完成整个流程控制处理，支持每一步报错的重做功能。进程主控每次取控制表步骤号，调用每一步子主控函数，子主控函数 fork 十几个进程，然后进行功能函数处理，子主控函数等待各子进程退出，完成该步子交易处理后，进程主控修改控制表步骤号并继续完成下一步。

在以前 OLTP 的项目中，因为是查找少量数据，where 子句上建索引可大 提高查询性能，但这条规律并不适用于 OLAP 项目。在并发分表时，由于各表都对客户号建了索引，分表时系统默认走索引，如把客户信息表分成 17 个表需要一个小时，但项目后期一次 然机会，我使用数据库伪指令（SELECT --+FULL），进行全表扫描，结果客户信息表分表只用了 20 分钟。可见，实践出真知，思考出智慧，有时 试一下新思路可能会有新的收获。

在数据库使用方法上，大表一般重新由小表并发生成，然后建索引；锁方式设置为 读，表数据更新后进行更新表的统计信息。处理完成后把原客户信息表、 献度表等重命名，并发重新生成新的客户信息表、 献度等表，生成新表后并发装载数据，然后再重建索引。

应用要求 fork 进程数必须完成规定个数，一个报错要杀掉其他进程，主进程要等待该步所有子进程功能完成才能进行下一步程序。下面的 demo 代码为该月末程序简化的多进程调度部分。

（2）下面是多进程并发处理的 demo 程序

mufork.c 源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#define ProNU 3 /** 进程数 ***/
int proc_id[ProNU] ; /**存放子进程的 id 号**/
int process()
{
    sleep(30) ; /**便于查看进程数*/
    printf("process success pid=%d\n",getpid()) ;
    return 0 ;
}
/*****
*      Function Name: cif_wait_proc( )
*      Description:   等待所有子进程退出
*      Return:        0 :    成功
*                    -1 :    失败
*****/
int cif_wait_proc(int *count, int proc_id[])
{
```

```

int status, i, j, pid ;
/* 父进程等待所有子进程完成 */
i = *count;
if ( *count < ProNU-1 ) /**子进程数未达规定数杀死所有子进程**/
{
    fprintf( stderr, "fork 失败" ) ;
    for( j = 0; j< *count; j++ )
    {
        kill(proc_id[j],9) ;
        sleep( 1 ) ;
        printf("fork 失败  " ) ;
        return -1 ;
    }
}
for( j = 0; j< *count; j++ )
{
    printf("proc[%d]=%ld\n",j, proc_id[j]) ;
}
/**子进程达到预计数等待所有子进程退出***/
while ( i ) {
    pid = wait( &status ) ;
    if ( pid < 0 ){
        continue;
    }
    for( j = 0; j< *count; j++ ) {
        /* 如果找到相应的子进程记录, 进行出错处理 */
        if ( pid == proc_id[j] ) {
            i--;
            if ( WIFSIGNALED( status ) ) {
                fprintf( stderr, "子进程[%d]被信号[%d]终止", pid, WTERMSIG
( status ) );

                for( j = 0; j< *count; j++ )
                {
                    kill(proc_id[j],9) ;
                    sleep( 1 ) ;
                }
                return -1;
            }
            else if ( WIFSTOPPED( status ) ) {
                fprintf( stderr, "子进程[%d]终止,终止信号[%d]",pid,WSTOPSIG
( status ) );

                for( j = 0; j< *count; j++ )
                {
                    kill(proc_id[j],9) ;
                    sleep( 1 ) ;
                }
                return -1;
            }
        }
    }
}
return 0 ;
}
int main()
{

```



```

int ret ,  paranu ,  dbsnu ,  minnul ,  maxnul ;
memset( proc_id, 0x00, sizeof(proc_id) ) ;
/*根据定制结果产生规定的进程数*/
for( paranu = 0 ; paranu < ProNU ; paranu++ ){
if( (proc_id[paranu] = fork()) == 0 ){
    ret = process();
    if( ret == -1 ){
        fprintf( stderr, "子进程出错,序号[%d]!\n", paranu) ;
        kill( getpid() , 9 );
    }
    exit( 0 ) ;
}
else if( proc_id[paranu] < 0 ){
    fprintf( stderr, "Fork 子进程出错,序号[%d]!\n", paranu) ;
    break ;
}
}
ret = cif_wait_proc(&paranu ,proc_id ) ;
if ( ret == -1 )
{
    fprintf(stderr,"子进程出错,序号[%d]!\n",paranu);
    return -1 ;
}
return 0;
}

```

编译 gcc mufork.c -o mufork。

./mufork, 可以用 ps -ef |grep mufork 查看进程数。

```

proc[0]=6923
proc[1]=6924
proc[2]=6925
process success pid=6923
process success pid=6924
process success pid=6925

```

第 13 章

Linux 线程编程

进程在各自独立的地址空间中运行。以前服务器端提供服务多采用多进程机制，而多进程机制需要 `fork` 子进程，子进程 `fork` 后需要单独的堆栈空间和系统资源，消耗系统开销和资源较大。而进程之间共享数据需要用进程间通信机制，增加了编程难度。现在较多的服务器端程序采用多线程机制，这种机制消耗资源少，也便于线程间共享数据，线程也有单独的堆栈空间，但消耗的时空成本比进程少许多。在一个进程的地址空间中执行多个线程，多线程同样可以执行多个控制流程，但其共享进程系统资源和全局数据。

13.1 线程简要说明

1. 线程概念

线程 (thread)，有时被称为轻量级进程，是程序执行流的最小单元。一个标准的线程由线程 ID、当前指令指针 (PC)、寄存器集合和堆栈组成。另外，线程是进程中的一个实体，是被系统独立调度和分派的基本单位，线程自己不拥有系统资源，只拥有少量在运行中必不可少的资源，但它可与同属一个进程的其他线程共享进程所拥有的全部资源。一个线程可以创建和撤销另一个线程，同一进程中的多个线程之间可以并发执行。由于线程之间的相互制约，致使线程在运行中呈现出间断性。线程也有就绪、阻塞和运行三种基本状态。

线程是程序中一个单一顺序控制流程，在单个程序中同时运行多个线程完成不同的工作，称为多线程。

2. 线程与进程

线程和进程的区别在于，子进程和父进程有不同的数据空间，而多个线程则共享数据空间，每个线程有自己的执行堆栈和程序计数器为其执行上下文。多线程主要是为了节约 CPU 时间和提高程序效率，线程的运行需要使用计算机的内存资源和 CPU。

通常在一个进程中可以包含若干个线程，它们可以利用进程所拥有的资源。在引入线程的操作系统中，通常都是把进程作为分配资源的基本单位，而把线程作为独立运行和独立调度的基本

单位。由于线程比进程更小，基本上不拥有系统资源，故对它的调度所付出的开销就会小得多，能更高效提高系统内多个程序间并发执行程度，从而显著提高系统资源的利用率和吞吐量。

3. 线程周期

线程生命周期由新建、就绪、运行、阻塞、死亡五部分组成。

4. 线程的好处

创建一个新线程花费的时间少，两个线程的切换时间少。

由于同一个进程内的线程共享内存和文件，所以线程之间互相通信必须调用内核。

线程能独立执行，能充分利用和发挥处理机与外围设备并行工作的能力。

5. 线程调度

当有线程进入了就绪状态，需要有线程调度程序来决定何时执行，根据优先级来调度。

6. 线程组

每个线程都是一个线程组的一个成员，线程组把多个线程集成一个对象，通过线程组，可以同时对其中的多个线程进行操作。在生成线程时，必须将线程放在指定的线程组，也可以放在默认的线程组中，默认的就是生成该线程所在的线程组。一旦一个线程加入了某个线程组，就不能被移出这个组。

7. 线程工作原理

线程是进程中的实体，一个进程可以拥有多个线程，一个线程必须有一个父进程。线程不拥有系统资源，只拥有运行需要的一些数据结构，它与父进程的其他线程共享该进程所拥有的全部资源。线程可以创建和撤销线程，从而实现程序的并发执行。

在多中央处理器的系统里，不同线程可以同时在不同的中央处理器上运行，甚至当它们属于同一个进程时也是如此。大多数支持多处理器的操作系统都提供编程接口来让进程可以控制自己的线程与各处理器之间的关联度。

有时候，线程也称为轻量级进程。像进程一样，线程在程序中有独立的、并发的执行路径，每个线程都有它自己的堆栈、自己的程序计数器和自己的局部变量。但是它们共享全局数据区、文件描述符和其他每个进程应有的状态。

进程可以支持多个线程，它们看似同时执行，但互相之间并不同步。一个进程中的多个线程共享相同的内存地址空间，这就意味着它们可以访问相同的变量和对象，而且它们从同一堆中分配对象。尽管这让线程之间共享信息变得更容易，但必须小心，确保它们不会妨碍同一进程里的其他线程。

编写有效使用线程的复杂程序并不十分容易，因为有多线程共存在相同的内存空间中并共享相同的变量，所以必须小心，确保线程不会互相干扰。

8. 线程资源

main 函数和信号处理函数是同一个进程地址空间中的多个控制流程，多线程也是如此，但是比信号处理函数更加灵活。信号处理函数的控制流程只是在信号递达时产生，在处理完信号之后就结束，而多线程的控制流程可以长期并存，操作系统会在各线程之间调度和切换，就像在多个进程之间调度和切换一样。由于同一进程的多个线程共享同一地址空间，因此文本段、数据段都是共享的。如果定义一个函数，在各线程中都可以调用，如果定义一个全局变量，在各线程中都可以访问到，除此之外，各线程还共享以下进程资源和环境：

- 文件描述符表；
- 每种信号的处理方式（SIG_IGN、SIG_DFL 或者自定义信号处理函数）；
- 当前工作目录；
- 用户 ID 和组 ID；
- 线程的全局变量。

但有些资源是每个线程各有一份的：

- 线程 ID；
- 上下文、包括各种寄存器的值、程序计数器和栈指针；
- 栈空间；
- errno 变量；
- 信号屏蔽字；
- 调度优先级。

在 Linux 上，线程函数位于 libpthread 共享库中，因此在编译时要加上 -lpthread 选项。

13.2 线程主要函数

1. 线程函数列表

表 13-1 列出了线程主要函数的名称及其简要说明。

表 13-1 线程主要函数列表

序 号	函数说明	函数名称
1	创建线程	pthread_create
2	等待线程结束	pthread_join
3	分离线程	pthread_detach
4	创建线程键	pthread_key_create

续表

序 号	函数说明	函数名称
5	删除线程键	pthread_key_delete
6	设置线程数据	pthread_setspecific
7	获取线程数据	pthread_getspecific
8	获取线程标识符	pthread_self
9	比较线程	pthread_equal
10	一次执行	pthread_once
11	出让执行权	sched_yield
12	修改优先级	pthread_setschedparam
13	获取优先级	pthread_getschedparam
14	发送信号	pthread_kill
15	设置线程掩码	pthread_sigmask
16	终止线程	pthread_exit
17	退出线程	pthread_cancel
18	允许/禁止退出线程	pthread_setcancelstate
19	设置退出类型	pthread_setcanceltype
20	创建退出点	pthread_testcancel
21	压入善后处理函数	pthread_cleanup_push
22	弹出善后处理函数	pthread_cleanup_pop

2. 线程函数具体说明

（1）创建线程 pthread_create

```
int pthread_create(pthread_t *tid, const pthread_attr_t *tattr, void
*(*start_routine)(void *), void *arg);
```

返回值：函数成功返回 0，任何其他返回值都表示错误。

函数说明：创建一个线程。

函数具体说明如下：

- ① 参数 tattr 中含有初始化线程所需要的属性，start_routine 是线程入口函数的地址，当 start_routine 返回时，相应的线程就结束了。
- ② 当函数成功时，线程标识符保存在参数 tid 指向的内存中。
- ③ 如果不指定属性对象，将其置为 NULL，则创建一个默认的线程，其属性为非绑定的、未分离的、有一个默认大小的堆栈，具有和父线程一样的优先级。
- ④ 在创建子线程时，传给子线程的输入参数最好是由 malloc() 函数返回的指针或指向全局变量的指针。

（2）等待线程结束 pthread_join

```
int pthread_join(pthread_t tid, void **status);
```

返回值：函数成功返回 0，任何其他返回值都表示错误。

函数说明：等待一个线程结束。

函数具体说明如下：

① 该函数阻塞调用线程，直到参数 `tid` 指定的线程结束。`tid` 指定的线程必须在当前进程中，同时，`tid` 指定的线程必须是非分离的。

② 不能有多个线程等待同一个线程终止。如果出现这种情况，一个线程将成功返回，别的线程将返回错误 `ESRCH`。

③ 如果参数 `status` 不为 `NULL`，则将线程退出状态放在 `status` 指向的内存中。

(3) 分离线程 `pthread_detach`

```
int pthread_detach(pthread_t tid);
```

返回值：函数成功返回 0，任何其他返回值都表示错误。

函数说明：将非分离的线程设置为分离线程，即通知线程库在指定的线程终止时，回收线程占用的内存等资源，在一个线程上使用多次 `pthread_detach` 的结果是不可预见的。

(4) 创建线程键 `pthread_key_create`

```
int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));
```

返回值：函数成功返回 0，任何其他返回值都表示错误。

函数说明：创建线程键。

函数具体说明如下：

① 在进程中分配一个键值，这个键被用来表示一个线程数据项，这个键对进程中所有的线程都是可见的。刚创建线程数据键时，在所有线程中和这个键相关联的值都是 `NULL`。

函数成功返回后，分配的键放在 `key` 参数指向的内存中，必须保证 `key` 参数指向的内存区的有效性。

② 如果指定了解析函数 `destructor`，那么当线程结束时，会将非空的值绑定在这个键上时，系统将调用 `destructor` 函数，参数就是相关线程与这个键绑定的值。绑定在这个键上的内存块可由 `destructor` 函数释放。

(5) 删除线程键 `pthread_key_delete`

```
int pthread_key_delete(pthread_key_t key);
```

返回值：函数成功返回 0，任何其他返回值都表示错误。

函数说明：删除线程数据键。这个键占用的内存将被释放，该键再被引用将返回错误。在调用该函数之前，程序必须释放和本线程相关联的资源，该函数不会引发线程数据键的解析函数。

(6) 设置线程数据 `pthread_setspecific`

```
int pthread_setspecific(pthread_key_t key, const void *value);
```

返回值：函数成功返回 0，任何其他返回值都表示错误。

函数说明：设置和某个线程数据键绑定在一起的线程专用数据（一般是指针）。

函数具体说明：函数不会释放原来绑定在键上的内存，给一个键值绑定新的指针时，必须释放原指针指向的内存，否则会发生内存泄漏。

(7) 获取线程数据 `pthread_getspecific`

```
void pthread_getspecific(pthread_key_t key, void **value);
```

返回值：无返回，出错时 `value` 指向 `NULL`。

函数说明：获取绑定在线程数据键上的值，并在指定的位置存储取来的值。

(8) 获取线程标识符 `pthread_self`

```
pthread_t pthread_self(void);
```

返回值：返回当前线程的标识符。

(9) 比较线程 `pthread_equal`

```
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

返回值：如果 `tid1` 和 `tid2` 相同，函数返回一个非 0 值，否则返回 0。如果 `tid1` 或 `tid2` 中任何一个是非法值，则返回将是不可预料的。

(10) 一次执行 `pthread_once`

```
int pthread_once(pthread_once_t *once_control, void (*init_routine)(void));
```

返回值：函数成功返回 0，任何其他返回值都表示错误。

函数说明：一次执行函数公共部分。

函数具体说明如下：

① 此函数用来调用初始化函数，如果已经有线程通过 `pthread_once` 调用过这个初始化函数一次，那么以后通过 `pthread_once` 函数再调用这个初始化函数将无效。

② 参数 `once_control` 决定了相应的初始化函数是否被调用过，它一般这样使用：`[static] pthread_once_t once_control = PTHREAD_ONCE_INIT`。

(11) 出让执行权 `sched_yield`

```
int sched_yield(void);
```

返回值：函数成功返回 0，-1 表示错误。

函数说明：把当前线程的执行权（即对处理器的控制权）出让给另一个有相同或更高优先级的线程。

（12）修改优先级 `pthread_setschedparam`

```
int pthread_setschedparam(pthread_t tid, int policy, const struct sched_param *param);
```

返回值：函数成功返回 0，任何其他返回值都表示错误。

函数说明：修改线程的优先级。

（13）获取优先级 `pthread_getschedparam`

```
int pthread_getschedparam(pthread_t tid, int policy, struct schedparam *param);
```

返回值：函数成功返回 0，任何其他返回值都表示错误。

函数说明：获取线程的优先级。

（14）发送信号 `pthread_kill`

```
int pthread_kill(pthread_t tid, int sig);
```

返回值：函数成功返回 0，任何其他返回值都表示错误。

函数说明：发送信号到线程。

函数具体说明如下：

- ① 向 `tid` 指定的线程发送一个信号，`tid` 指定的线程必须和当前线程在同一个进程中。
- ② 当 `sig` 参数为 0 时，函数将进行错误检查，不发送信号，这常常用来检查 `tid` 的合法性。

（15）设置线程掩码 `pthread_sigmask`

```
int pthread_sigmask(int how, const sigset_t *new, sigset_t *old);
```

返回值：函数成功返回 0，任何其他返回值都表示错误。

函数说明：改变或检验当前线程的信号掩码。

函数具体说明如下：

- ① 参数 `how` 表示对当前信号掩码进行什么操作，有 `SIG_BLOCK`、`SIG_UNBLOCK`、`SIG_SETMASK` 三种值。
- ② 当参数 `new` 为 `NULL` 时，不论 `how` 的值是什么，当前线程的信号掩码都不会改变。旧的信号掩码保存在参数 `old` 指向的内存中，当 `old` 不为 `NULL` 时。

（16）终止线程 `pthread_exit`

```
void pthread_exit(void *status);
```


函数说明：终止当前线程。

函数具体说明如下：

① 终止当前线程，所有绑定在线程数据键上的内存将被释放。如果当前线程是非分离的，那么这个线程的退出代码将被保留，直到其他线程用 `pthread_join` 来等待当前线程的终止。如果当前线程是分离的，`status` 将被忽略，线程标识符将被立即回收。

② 若 `status` 不为 `NULL`，线程的退出代码被置为 `status` 参数指向的值。

(17) 退出线程 `pthread_cancel`

```
int pthread_cancel(pthread_t thread);
```

返回值：函数成功返回 0，任何其他返回值都表示错误。

函数说明：退出一个线程。

(18) 允许/禁止退出线程 `pthread_setcancelstate`

```
int pthread_setcancelstate(int state, int *oldstate);
```

返回值：函数成功返回 0，任何其他返回值都表示错误。

函数说明：参数 `state` 取值为 `PTHREAD_CANCEL_ENABLE` 或 `PTHREAD_CANCEL_DISABLE`。

(19) 设置退出类型 `pthread_setcanceltype`

```
int pthread_setcanceltype(int type, int *oldtype);
```

返回值：函数成功返回 0，任何其他返回值都表示错误。

函数具体说明如下：

① 将线程退出类型设置为延迟类型或异步类型。参数 `type` 的取值为 `PTHREAD_CANCEL_DEFERRED` 或 `PTHREAD_CANCEL_ASYNCYNONS`。

② 当一个线程被创建后，默认值是延迟类型。在异步方式下，线程可以在执行的任何时候退出。

(20) 创建退出点 `pthread_testcancel`

```
void pthread_testcancel(void);
```

返回值：无返回值。

函数具体说明如下：

① 设置线程的退出点。只有当线程的退出状态是允许退出的，而且线程的退出类型是延迟时，调用该函数才有效。如果调用时，线程的退出状态是禁止的，则该调用不起作用。

② 小心使用该函数，只有在能够安全退出的地方才能够设置退出点。

(21) 压入善后处理函数

```
void pthread_cleanup_push(void (*routine)(void *), void *args);
```

函数说明：将一个善后处理函数压入善后处理函数堆栈。

(22) 弹出善后处理函数

```
void pthread_cleanup_pop(int execute);
```

函数具体说明如下：

① 从善后处理函数堆栈中弹出一个善后处理函数。如果参数 `execute` 非 0，则执行弹出的函数；如果参数为 0，则不执行弹出函数。

② 如果一个线程显式或隐式地调用 `pthread_exit()` 函数或线程接受了退出请求，线程库实际上将会以非 0 参数调用 `pthread_cleanup_pop` 函数。

13.3 线程编程

13.3.1 线程创建

(1) 创建线程

POSIX 通过 `pthread_create()` 函数创建线程，函数原型如下：

```
#include <pthread.h>
int pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict
attr,
void *(*start_routine)(void*), void *restrict arg)
```

函数具体说明如下：

① 返回值：成功返回 0，失败返回错误号。系统函数一般都是成功返回 0，失败返回 -1，而错误号保存在全局变量 `errno` 中。而 `pthread` 库的函数都是通过返回值返回错误号，虽然每个线程也都有一个 `errno`，但这是为了兼容其他函数接口而提供的，`pthread` 库本身并不使用它，通过返回值返回错误码更加清晰。

② 在一个线程中调用 `pthread_create()` 创建新的线程后，当前线程从 `pthread_create()` 返回继续往下执行，而新的线程执行代码由 `start_routine` 指针函数决定。`start_routine` 指针函数接收一个参数，是通过 `pthread_create` 函数的 `arg` 参数传递给它的，该参数的类型为 `void *`，这个指针按什么类型解释由调用者自己定义。`start_routine` 的返回值类型也是 `void *`，这个指针的类型同样由调用者自己定义。`start_routine` 返回时，这个线程就退出了，其他线程可以调用 `pthread_join` 得到 `start_routine` 的返回值，类似于父进程调用 `wait()` 得到子进程的退出状态。

③ `pthread_create` 函数成功返回后，新创建的线程 ID 被填写到 `thread` 参数所指向的

内存单元。进程 ID 的类型是 `pid_t`，每个进程 ID 在整个系统中是唯一的，调用 `getpid()` 可以获得当前进程 ID，是一个正整数值。线程 ID 的类型是 `thread_t`，它只在当前进程中保证是唯一的，在不同系统中 `thread_t` 这个类型有不同的实现，它可能是一个整数值，也可能是一个结构体，甚至可能是一个地址，所以不能简单地当成整数而使用 `printf` 打印，调用 `pthread_self()` 可以获得当前线程 ID。

④ `attr` 参数是一个结构指针，结构中的元素分别对应着新线程的运行属性，主要包括以下几项：

__detachstate: 表示新线程是否与进程中其他线程脱离同步，如果设置，则新线程不能用 `pthread_join()` 来同步，且在退出时自行释放所占用的资源，默认为 `PTHREAD_CREATE_JOINABLE` 状态。这个属性也可以在线程创建并运行以后，用 `pthread_detach()` 来设置。但一旦设置为 `PTHREAD_CREATE_DETACH` 状态（不论是创建时设置还是运行时设置），则不能再恢复到 `PTHREAD_CREATE_JOINABLE` 状态。

__schedpolicy: 表示新线程的调度策略，主要包括 `SCHED_OTHER`（正常、非实时）、`SCHED_RR`（实时、轮转法）和 `SCHED_FIFO`（实时、先入先出）三种，默认为 `SCHED_OTHER`，后两种调度策略仅对超级用户有效，运行时可以用 `pthread_setschedparam()` 来改变。

__schedparam: 一个 `struct sched_param` 结构，目前仅有一个 `sched_priority` 整型变量表示线程的运行优先级。这个参数仅当调度策略为实时（即 `SCHED_RR` 或 `SCHED_FIFO`）时才有效，并可以在运行时通过 `pthread_setschedparam()` 来改变，默认为 0。

__inheritsched: 有两种值可供选择，`PTHREAD_EXPLICIT_SCHED` 和 `PTHREAD_INHERIT_SCHED`，前者表示新线程使用显式调度策略和调度参数（即 `attr` 中的值），而后者表示继承调用者线程的值，默认为 `PTHREAD_EXPLICIT_SCHED`。

__scope: 表示线程间竞争 CPU 的范围，也就是说线程优先级的有效范围。POSIX 标准中定义了两个值，即 `PTHREAD_SCOPE_SYSTEM` 和 `PTHREAD_SCOPE_PROCESS`，前者表示与系统中所有线程一起竞争 CPU 时间，后者表示仅与同进程中的线程竞争 CPU。目前 Linux 线程仅实现了 `PTHREAD_SCOPE_SYSTEM` 一值。

`pthread_attr_t` 结构中还有一些值，但不使用 `pthread_create()` 来设置。为了设置这些属性，POSIX 定义了一系列属性设置函数，包括 `pthread_attr_init()`、`pthread_attr_destroy()` 和与各个属性相关的 `get/set` 函数。

（2）线程编程创建示例

`pthread_create.c` 源代码如下：

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
pthread_t ntid;
```

```
void printids(const char *s)
{
    pid_t    pid;
    pthread_t tid;
    pid = getpid();
    tid = pthread_self();
    printf("%s pid %u tid %u (0x%x)\n", s, (unsigned int)pid,
          (unsigned int)tid, (unsigned int)tid);
}

void *thr_fn(void *arg)
{
    printids(arg);
    return NULL;
}

int main(void)
{
    int err;
    err = pthread_create(&tid, NULL, thr_fn, "new thread: ");
    if (err != 0) {
        fprintf(stderr, "can't create thread: %s\n", strerror(err));
        exit(1);
    }
    printids("main thread:");
    sleep(1);
    return 0;
}
```

编译 `gcc pthread_create.c -o pthread_create -lpthread`。

执行 `./pthread_create`，执行结果如下：

```
new thread: pid 12686 tid 3084622736 (0xb7db9b90)
main thread: pid 12686 tid 3084625584 (0xb7dba6b0)
```

在 Linux 上，调用 `pthread_self()` 可以得到线程 ID 号。

由于 `pthread_create` 的错误码不保存在 `errno` 中，因此不能直接用 `perror()` 打印错误信息，可以先用 `strerror()` 把错误码转换成错误信息再打印。

13.3.2 终止线程

如果需要只终止某个线程而不终止整个进程，可以有三种方法：

① 从线程函数 `return` 返回，这种方法对主线程不适用，因为从 `main` 函数 `return` 返回相当于调用 `exit` 退出。

② 一个线程可以调用 `pthread_cancel` 函数终止同一进程中的另一个线程。

③ 线程调用 `pthread_exit` 函数会终止自己，然后调用 `pthread_join` 函数等待线程结束。

(1) `pthread_exit` 函数具体说明

```
#include <pthread.h>
void pthread_exit(void *value_ptr)
```

`value_ptr` 是 `void *` 类型，和线程函数返回值用法一样，其他线程可以调用 `pthread_join` 获得这个指针。

需要注意，`pthread_exit` 或者 `return` 返回的指针所指向的内存单元必须是全局的或者是用 `malloc` 函数分配的，不能在线程函数栈上分配，因为当其他线程得到这个返回指针时，线程函数可能已经退出了。

(2) pthread_join 函数具体说明

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **value_ptr)
```

返回值：成功返回 0，失败返回错误号。

调用该函数的线程将挂起等待，直到线程 ID 为 `thread` 的线程终止。`thread` 线程以不同的方法终止，通过 `pthread_join` 得到的终止状态是不同的，总结如下：

① 如果 `thread` 线程通过 `return` 返回，`value_ptr` 所指向的单元里存放的是 `thread` 线程函数的返回值。

② 如果 `thread` 线程被别的线程调用 `pthread_cancel` 异常终止掉，`value_ptr` 所指向的单元里存放的是常数 `PTHREAD_CANCELED`，其值为宏定义，可以在头文件 `pthread.h` 中找到它的定义 `#define PTHREAD_CANCELED ((void *) -1)`。

③ 如果 `thread` 线程是自己调用 `pthread_exit` 终止的，`value_ptr` 所指向的单元存放的是传给 `pthread_exit` 的参数。

④ 如果对 `thread` 线程的终止状态不感兴趣，可以传 `NULL` 给 `value_ptr` 参数。

(3) 终止线程编程示例

`pthread_exit.c` 源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
void *thr_fn1(void *arg)
{
    printf("thread 1 returning\n");
    return (void *)1;
}
void *thr_fn2(void *arg)
{
    printf("thread 2 exiting\n");
    pthread_exit((void *)2);
}
void *thr_fn3(void *arg)
{

```

```
while(1) {
    printf("thread 3 writing\n");
    sleep(1);
}
}
int main(void)
{
    pthread_t  tid;
    void      *tret;
    pthread_create(&tid, NULL, thr_fn1, NULL);
    pthread_join(tid, &tret);
    printf("thread 1 exit code %d\n", (int)tret);
    pthread_create(&tid, NULL, thr_fn2, NULL);
    pthread_join(tid, &tret);
    printf("thread 2 exit code %d\n", (int)tret);
    pthread_create(&tid, NULL, thr_fn3, NULL);
    sleep(3);
    pthread_cancel(tid);
    pthread_join(tid, &tret);
    printf("thread 3 exit code %d\n", (int)tret);
    return 0;
}
```

编译 `gcc pthread_exit.c -o pthread_exit -lpthread`。

执行 `./pthread_exit`，执行结果如下：

```
thread 1 returning
thread 1 exit code 1
thread 2 exiting
thread 2 exit code 2
thread 3 writing
thread 3 writing
thread 3 writing
thread 3 exit code -1
```

一般情况下，线程终止后，其终止状态一直保留到其他线程调用 `pthread_join` 获取它的状态为止。但是线程也可以被置为 `detach` 状态，这样，线程一旦终止就立刻回收它占用的所有资源，而不保留终止状态。不能对一个已经处于 `detach` 状态的线程调用 `pthread_join` 函数，这样的调用将返回 `EINVAL` 错误。对一个尚未 `detach` 的线程调用 `pthread_join` 函数或 `pthread_detach` 函数都可以把该线程置为 `detach` 状态，也就是说，不能对同一线程调用两次 `pthread_join` 函数，或者如果已经对一个线程调用了 `pthread_detach` 函数就不能再调用 `pthread_join` 函数了。

13.3.3 线程互斥

(1) 互斥量

互斥量是一种锁，在访问共享资源时对其加锁，在结束访问时释放锁。这样可以保证在任意时间内，只有一个线程处于临界区内，任何要进入临界区的线程都要对锁进行测试，如果该锁已经被某一线程所持有，则测试线程会被阻塞，直到该锁被释放，线程会重复上述过程。

在线程没有释放锁之前，所有试图进入临界区的线程都被阻塞，形成一个阻塞线程队列，如图 13-1 所示。

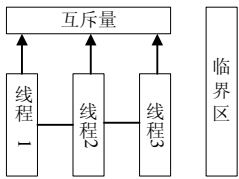


图 13-1 线程互斥图

(2) 申请一个互斥锁变量

```
pthread_mutex_t mutex; //申请一个互斥锁
```

可以利用全局变量声明多个互斥量，也可以利用 malloc 函数在堆上申请多个互斥锁变量。利用 pthread_mutex_init() 初始化互斥锁变量，其函数原型如下：

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)
```

第一个参数 mutex 是之前声明的互斥量，第二个参数为该互斥量的属性。互斥量分为下面四种类型：

- ① PTHREAD_MUTEX_TIMED_NP，这是默认值，也就是普通锁。当一个线程加锁以后，其余请求锁的线程将形成一个等待队列，并在解锁后按优先级获得锁。这种锁策略保证了资源分配的公平性。
- ② PTHREAD_MUTEX_RECURSIVE_NP，嵌套锁，允许同一个线程对同一个锁成功获得多次，并通过多次 unlock 解锁。如果是不同线程请求，则在加锁线程解锁时重新竞争。
- ③ PTHREAD_MUTEX_ERRORCHECK_NP，检错锁，如果同一个线程请求同一个锁，则返回 EDEADLK 错误，否则与 PTHREAD_MUTEX_TIMED_NP 类型动作相同。这样就保证当不允许多次加锁时，不会出现最简单情况下的死锁。
- ④ PTHREAD_MUTEX_ADAPTIVE_NP，适应锁，动作最简单的锁类型，仅等待解锁后重新竞争。

以下语句可以做到将一个互斥锁快速初始化。

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER
```

(3) 注销一个互斥锁

pthread_mutex_destroy() 用于注销一个互斥锁，函数原型如下：

```
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

注销一个互斥锁即意味着没有线程再占有该锁，销毁后该锁处于开放状态。

(4) 上锁

在创建该互斥量之后，便可以给互斥量上锁，其函数原型如下：

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

该函数用来给互斥量上锁，互斥量一旦被上锁后，其他线程如果想给该互斥量上锁，那么就

会阻塞在这个操作上。如果在此之前，该互斥量已经被其他线程上锁，那么该操作将会一直阻塞在这个地方，直到获得该锁为止。

(5) 解锁

下面的函数可以用来给互斥量解锁，这样其他等待该锁的线程才有机会获得该锁，否则其他线程将会永远阻塞。其函数原型如下：

```
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

(6) 非阻塞上锁

如果不想上锁碰到一直阻塞的情况，可以用非阻塞方式上锁，其函数原型如下：

```
int pthread_mutex_trylock(pthread_mutex_t *mutex)
```

如果此时互斥量没有被上锁，那么会返回 0，并对该互斥量上锁；如果互斥量已经被上锁，那么会立刻返回 EBUSY 错误。

(7) 线程编程互斥示例

pthread_mutex.c 源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NLOOP 3
int counter; /* incremented by threads */
pthread_mutex_t counter_mutex = PTHREAD_MUTEX_INITIALIZER;
void *doit(void *);
int main(int argc, char **argv)
{
    pthread_t tidA, tidB;
    pthread_create(&tidA, NULL, doit, NULL);
    pthread_create(&tidB, NULL, doit, NULL);
    /* wait for both threads to terminate */
    pthread_join(tidA, NULL);
    pthread_join(tidB, NULL);
    return 0;
}
void *doit(void *vptr)
{
    int i, val;
    sleep(3); //防止线程过早退出
    for (i = 0; i < NLOOP; i++) {
        pthread_mutex_lock(&counter_mutex);
        val = counter;
        printf("%u: %d\n", (unsigned int)pthread_self(), val + 1);
        counter = val + 1;
        pthread_mutex_unlock(&counter_mutex);
    }
    return NULL;
}
```


编译 `gcc pthread_mutex.c -o pthread_mutex -lpthread`。

执行 `./pthread_mutex`，执行结果如下：

```
3084413840: 1
3084413840: 2
3084413840: 3
3076021136: 4
3076021136: 5
3076021136: 6
```

13.3.4 线程同步

(1) 条件变量说明

与互斥锁不同，同步条件变量是用来等待而不是用来上锁的。条件变量用来自动阻塞一个线程，直到某特殊情况发生为止，通常条件变量和互斥锁同时使用。

条件变量可以使线程睡眠，等待某种条件出现，条件变量是线程间共享全局变量进行同步的一种机制。

条件的检测是在互斥锁的保护下进行的，如果一个条件为假，一个线程自动阻塞，并释放等待状态改变的互斥锁；如果另一个线程改变了条件，它发信号给关联的条件变量，唤醒一个或多个等待它的线程，并重新获得互斥锁，重新评价条件。

条件变量分为两部分——条件和变量。条件本身是由互斥量保护的，线程在改变条件状态前，先要锁住互斥量，它是利用线程间共享全局变量进行同步的一种机制。

(2) 条件变量初始化

条件变量采用的数据类型是 `pthread_cond_t`，在使用之前必须要进行初始化，初始化包括以下两种方式：

① 静态：可以把常量 `PTHREAD_COND_INITIALIZER` 赋值给静态分配的条件变量。

② 动态：调用 `pthread_cond_init` 函数动态创建条件变量。释放动态条件变量的内存空间之前，需用 `pthread_cond_destroy` 函数对其进行清理。

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *restrict cond, pthread_condattr_t
*restrict attr)
int pthread_cond_destroy(pthread_cond_t *cond)
```

这两个函数成功则返回 0，出错则返回错误编号。当 `pthread_cond_init` 函数的 `attr` 参数为 `NULL` 时，会创建一个默认属性的条件变量。

(3) 等待条件

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t
```

```
*restrict mutex);  
int pthread_cond_timedwait(pthread_cond_t *restrict cond, pthread_mutex_t  
*restrict mutex, const struct timespec *restrict timeout);
```

这两个函数成功则返回 0，出错则返回错误编号。这两个函数分别是阻塞等待条件和超时等待条件。当等待条件变为真，使用互斥量对条件进行保护，调用者把锁住的互斥量传递给函数。函数把调用线程放到等待条件的线程列表中，然后对互斥量解锁，这两个操作都是原子的。这样便关闭了条件检查和线程进入休眠状态等待条件改变这两个操作之间的时间通道，线程就不会错过条件的任何变化。当函数返回时，互斥量再次被锁住。

(4) 通知条件

```
#include <pthread.h>  
int pthread_cond_signal(pthread_cond_t *cond)  
int pthread_cond_broadcast(pthread_cond_t *cond)
```

这两个函数成功则返回 0，出错则返回错误编号。这两个函数用于通知线程条件已经满足，向线程或条件发送信号。必须注意的是，一定要在改变条件状态以后，再给线程发送信号。`pthread_cond_signal()` 激活一个等待该条件的线程，存在多个等待线程时，按入队顺序激活其中一个，而 `pthread_cond_broadcast()` 则激活所有等待线程。

(5) 条件变量与互斥锁的区别

互斥锁要么锁住，要么被解开，二值状态，类似二值信号量。

互斥锁是为了上锁而设计的，而条件变量是为了条件等待而设计的。

(6) 线程编程同步示例

`pthread_cond.c` 源代码如下：

```
#include <stdio.h>  
#include <pthread.h>  
pthread_mutex_t mutex;  
pthread_cond_t cond;  
void *thread1(void*arg)  
{  
    while(1){  
        printf("thread1 is running\n");  
        pthread_mutex_lock(&mutex);  
        pthread_cond_wait(&cond,&mutex);  
        printf("thread1 applied the condition\n");  
        pthread_mutex_unlock(&mutex);  
        sleep(3);  
    }  
}  
void *thread2(void*arg)  
{  
    while(1){  
        printf("thread2 is running\n");  
        pthread_mutex_lock(&mutex);  
        pthread_cond_wait(&cond,&mutex);
```

```
        printf("thread2 applied the condition\n");
        pthread_mutex_unlock(&mutex);
        sleep(1);
    }
}
int main()
{
    pthread_t thid1, thid2;
    printf("condition variable study!\n");
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);
    pthread_create(&thid1, NULL, (void*)thread1, NULL);
    pthread_create(&thid2, NULL, (void*)thread2, NULL);
    do{
        pthread_cond_signal(&cond);
    }while(1);
    pthread_exit(0);
    return 0;
}
```

编译 gcc pthread_cond.c -o pthread_cond -lpthread。

执行 ./pthread_cond, 执行结果如下:

```
ccondition variable study!
thread1 is running
thread2 is running
thread1 applied the condition
thread2 applied the condition
thread2 is running
thread2 applied the condition
thread2 is running
thread2 applied the condition
thread1 is running
```

第 14 章

Linux 进程间通信——管道与信号

管道和信号是进程间通信的两种机制。生活中水的管道是从一端流入，然后从另一端流出。与此类似，操作系统中的每一个管道有两个文件描述符，一个文件描述符用来读，另一个用来写。信号是一个软件中断，主要用于进程间异步事件的通知与进程控制。

14.1 进程间通信概述

1. 进程间通信的类型

进程间通信的类型有如下六种。

- 管道（pipe）和命名管道（FIFO）。
- 信号（signal）。
- 共享内存。
- 消息队列。
- 信号量。
- 套接字（socket）。

2. 进程间通信的目的

进程间通信的目的有如下五种。

- 数据传输：一个进程需要将它的数据发送给另一个进程。
- 共享数据：多个进程想要共享数据，一个进程对共享数据进行修改后，其他进程可以立刻看到。
- 通知事件：一个进程需要向另一个或一组进程发送消息，通知它（它们）发生了某种事件（如进程终止时要通知父进程）。

- 资源共享：多个进程之间共享同样的资源。为了做到这一点，需要内核提供锁和同步机制。
- 进程控制：有些进程希望完全控制另一个进程的执行（如 Debug 进程），此时，控制进程希望能够拦截另一个进程的所有陷入和异常，并能够及时知道它的状态改变。

14.2 管道

管道是 Linux 中最常用的进程间通信 IPC 机制。使用管道时，一个进程的输出可成为另外一个进程的输入。

当输入/输出的数据量特别大时，管道的这种 IPC 机制就非常有用。

在 Linux 中，通过将两个 file 结构指向同一个临时的 VFS 索引节点，这个 VFS 索引节点又指向同一个物理页而实现管道。

如图 14-1 所示为管道的内核实现图，每个 file 数据结构定义不同的文件操作地址，其中一个用来向管道中写入数据，另一个用来从管道中读出数据。

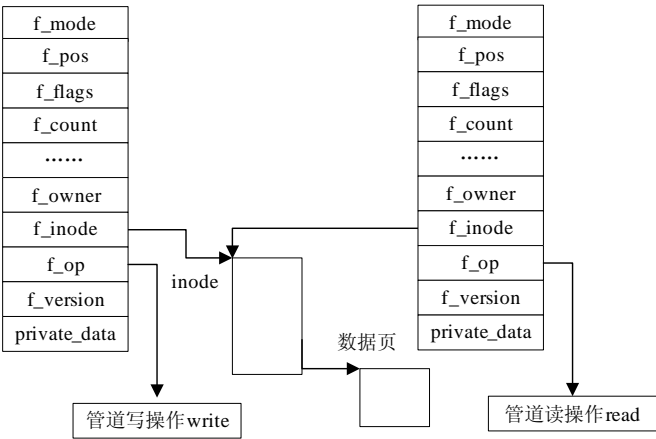


图 14-1 管道的内核实现图

当写进程向管道中写入时，它利用标准的库函数进行写操作，系统根据库函数传递的文件描述符可找到该文件的 file 结构。file 结构中指定了用来进行写操作的地址，于是，内核调用写函数向指定地址完成写操作。

写入函数在向内存中写入数据之前，首先必须检查 VFS 索引节点信息，同时满足如下条件时，才能进行实际的内存复制工作。

- 内存中有足够的空间可容纳要写入的所有数据。
- 内存没有被读程序锁定。

管道的读取过程和写入过程类似。

管道的打开模式为非阻塞的情况下，进程可以在没有数据或内存被锁定时立即返回错误信息，

而不是阻塞该进程。管道的打开模式为阻塞的情况下，进程可以在索引节点的等待队列中休眠等待写入进程写入数据。当所有的进程完成了管道操作之后，管道的索引节点被丢弃，而共享数据页也被释放。

上面讲述的管道类型也被称为“匿名管道”。匿名管道包括 `pipe` 管道和标准流管道。

Linux 还支持另外一种管道形式，称为命名管道（FIFO），这种管道的操作方式基于“先进先出”原理。

命名管道和匿名管道的数据结构以及操作极其类似，二者的主要区别在于：命名管道在使用之前就已经存在，用户可打开或关闭命名管道；而匿名管道只在操作时存在，因而是临时对象。

14.2.1 pipe 管道

1. pipe 函数原型

若要创建一个简单的管道，可以使用系统调用`pipe()`，它接受一个参数，也就是一个包括两个整数的数组。如果系统调用成功，此数组将包括管道使用的两个文件描述符，一个为读端，一个为写端。`pipe`管道是半双工的，数据只能向一个方向流动，需要双方通信时，就建立起两个管道。`pipe`管道只能用于父子进程或者兄弟进程之间（具有亲缘关系的进程）。

`pipe` 函数原型如下：

pipe（建立简单管道）	
所需头文件	#include <unistd.h>
函数说明	<code>pipe()</code> 会建立管道，并将文件描述符由参数 <code>filedes</code> 数组返回
函数原型	int pipe(int filedes[2])
函数传入值	<code>filedes</code> ：管道文件描述符， <code>filedes[0]</code> 为管道的读取端， <code>filedes[1]</code> 则为管道的写入端
函数返回值	成功：0
	出错：-1，错误原因存于 <code>errno</code> 中
错误代码	EMFILE：没有空闲的文件描述符 EMFILE：系统文件表已满 EFAULT：fd 数组无效
附加说明	管道主要用于父子进程间通信。实际上，通常先创建一个管道，再通过 <code>fork</code> 函数创建一个子进程。其实 <code>pipe</code> 的两个文件描述符指向相同的内存空间，只不过 <code>filedes[1]</code> 有写权限， <code>filedes[0]</code> 有读权限

2. pipe 管道原理测试范例

下列程序测试 `pipe` 管道的实现原理，`pipetest.c` 源代码如下：

```
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
void look_into_pipe() {
    int n;
```

```

int fd[2];
char line[1024];
struct stat buf;
if (pipe(fd) < 0) {          /*创建管道*/
    printf("pipe error.\n");
    return;
}
fstat(fd[0], &buf);
if (S_ISFIFO(buf.st_mode)) { /*S_ISFIFO 为测试此文件类型是否为管道文件*/
    printf("fd[0]: FIFO file type.\n");
}
printf("fd[0]: inode=%d\n", buf.st_ino);
fstat(fd[1], &buf);
if (S_ISFIFO(buf.st_mode)) {
    printf("fd[1]: FIFO file type.\n");
}
printf("fd[1]: inode=%d\n", buf.st_ino);
write(fd[1], "hello world.\n", 12);
n = read(fd[0], line, 512 );
write(STDOUT_FILENO, line, n);
n=write(fd[0], "HELLO WORLD.\n", 12); /*0 端只允许读, 1 端允许写, 这是便于测试*/
if ( -1==n )
{
    printf("\nwrite error\n") ;
}
}
int main(){
    look_into_pipe() ;
}

```

编译 gcc pipetest.c -o pipetest。

执行 ./pipetest, 执行结果如下:

```

fd[0]: FIFO file type.
fd[0]: inode=24962
fd[1]: FIFO file type.
fd[1]: inode=24962
hello world.
write error

```

3. pipe 管道典型应用范例

(1) pipe 管道典型应用程序

pipe 管道的典型应用是使用在父子通信的场合。下列程序是 pipe 管道典型应用程序范例, pipe.c 源代码如下:

```

#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

```

```
int main(){
    int pipe_fd[2];
    pid_t pid;
    char buf_r[100];
    char *p_wbuf;
    int r_num;

    memset(buf_r,0,sizeof(buf_r));
    //创建管道
    if(pipe(pipe_fd)<0){
        perror("pipe create error\n");
        return -1;
    }
    if((pid=fork())==0){//表示在子进程中
        //关闭管道写描述符,进行管道读操作
        printf("child pipe1=%d; pipe2=%d\n", pipe_fd[0], pipe_fd[1] );
        close(pipe_fd[1]);
        //从管道描述符中读取
        sleep(2) ;
        if((r_num=read(pipe_fd[0],buf_r,100))>0){
            printf("%d numbers read from the pipe, data is %s\n",r_num,buf_r);
        }
        close(pipe_fd[0]);
        exit(0);
    }
    else if(pid>0){//表示在父进程中,父进程写
        //关闭管道读描述符,进行管道写操作
        printf("parent pipe1=%d; pipe2=%d\n", pipe_fd[0], pipe_fd[1] );
        close(pipe_fd[0]);
        if(write(pipe_fd[1],"Hello",5)!=-1)
            printf("parent writel success!\n");
        if(write(pipe_fd[1]," Pipe",5)!=1)
            printf("parent write2 success!\n");
        close(pipe_fd[1]);
        sleep(3) ;
        waitpid(pid,NULL,0);
        exit(0);
    }else{
        perror("fork error");
        exit(-1);
    }
}
```

编译 gcc pipe.c -o pipe。

执行 ./pipe, 执行结果如下:

```
child pipe1=3; pipe2=4
parent pipe1=3; pipe2=4
parent writel success!
parent write2 success!
10 numbers read from the pipe, data is Hello Pipe
```

(2) pipe 管道实现图解

对于 pipe.c 程序, 其管道实现过程如图 14-2~图 14-4 所示。

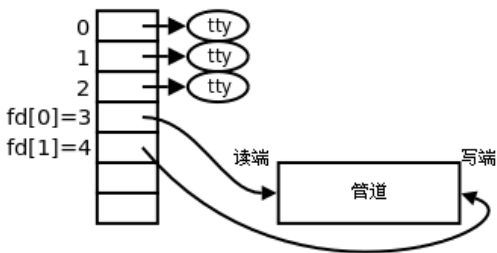


图 14-2 父进程创建的管道

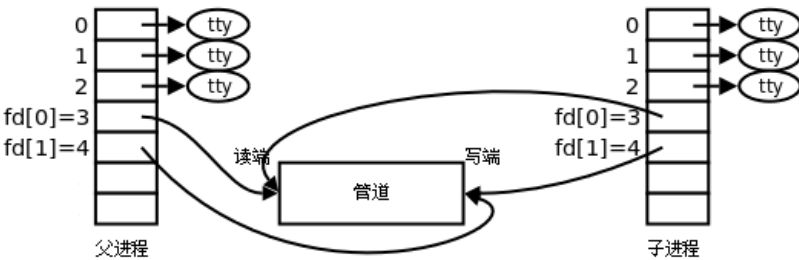


图 14-3 父进程调用 fork 产生子进程

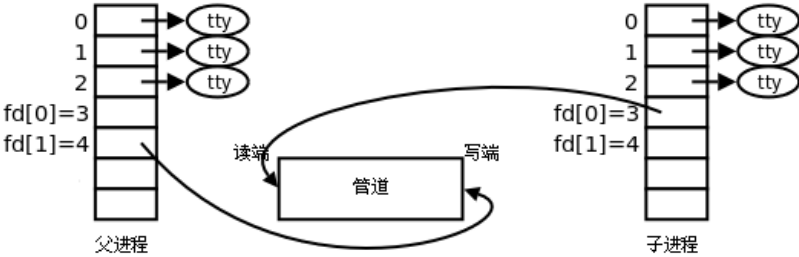


图 14-4 父进程关闭 fd[0]，子进程关闭 fd[1]

管道的概念来自实际生活中的管道，但读端和写端指向相同的 inode 地址，图 14-2～图 14-4 这样画是便于大家能形象地理解。由于进程的文件描述符 0 指向标准输入（键盘），1 指向标准输出（屏幕），2 指向标准错误（屏幕），所以进程可用的文件描述符从 3 开始。进程是通过文件描述符的操作，从而实现对文件描述符所指向文件的操作。

(3) pipe.c 程序的简要说明

对于 pipe.c 程序，简要说明如下：

父进程调用 fork() 产生子进程时，子进程几乎复制了父进程的所有资源，所以也复制了管道文件描述符。由于两个进程的管道文件描述符都指向同一个 inode 地址，所以能实现相互通信。

本程序实现的是父子进程单工通信，一个进程为读端，另一个进程为写端，所以写端需要关闭读文件描述符，读端需要关闭写文件描述符，以免误操作。一个进程在一个管道上完成又读又写功能，虽然在技术上能实现，但会造成复杂的程序控制流程和功能的不清晰。若要实现父子进程的双工通信，一般是通过建立两个管道来实现，双工通信的实现原理与单工通信类似。

14.2.2 标准流管道

像 Linux 系统中的文件操作是基于文件流标准 I/O 一样，管道的操作也支持文件流模式，这种管道称为标准流管道。标准流管道通过 `popen()` 创建一个管道，`popen()` 会调用 `fork()` 产生一个子进程，执行一个 Shell 以运行命令来开启一个进程，并把执行结果写入管道中，然后返回一个文件指针。程序通过文件指针可读取管道中的内容。使用 `popen()` 创建的标准流管道需要用 `pclose()` 进行关闭。

1. 标准流管道函数说明

`popen` 函数的函数原型及其具体说明如下。

popen（建立标准流管道）	
所需头文件	#include <stdio.h>
函数说明	<code>popen()</code> 会调用 <code>fork()</code> 产生子进程，然后从子进程中调用 <code>/bin/sh -c</code> 来执行参数 <code>command</code> 的指令。参数 <code>type</code> 可使用“r”代表读取，“w”代表写入。依照此 <code>type</code> 值， <code>popen()</code> 会建立管道连接到子进程的标准输出设备或标准输入设备，然后返回一个文件指针。随后进程便可利用此文件指针来读取子进程的输出设备或是写入子进程的标准输入设备中。此外，所有使用文件指针(FILE*)操作的函数也都可以使用，除 <code>fclose()</code> 以外
函数原型	FILE * popen (const char * command,const char * type)
函数传入值	command: 执行的指令
	type: “r” 代表读取，“w” 代表写入
函数返回值	成功: 文件流指针
	出错: NULL, 错误原因存于 <code>errno</code> 中
错误代码	EINVAL: 参数 <code>type</code> 不合法
注意事项	在编写具有 SUID/SGID 权限的程序时请尽量避免使用 <code>popen()</code> ， <code>popen()</code> 会继承环境变量，通过环境变量可能会造成系统安全的问题
附加说明	使用 <code>popen()</code> 创建的管道必须使用 <code>pclose()</code> 关闭。其实， <code>popen()</code> 、 <code>pclose()</code> 和标准文件输入/输出流中的 <code>fopen()</code> 、 <code>fclose()</code> 十分相似

`Pclose` 函数的原型及其具体说明如下。

pclose（关闭标准流管道）	
所需头文件	#include <stdio.h>
函数说明	<code>pclose()</code> 用来关闭由 <code>popen</code> 建立的管道及文件指针。参数 <code>stream</code> 是先前由 <code>popen()</code> 所返回的文件指针
函数原型	int pclose(FILE * stream)
函数传入值	stream: 文件流指针
函数返回值	成功: 子进程的结束状态
	失败: -1, 错误原因存于 <code>errno</code> 中
错误代码	ECHILD: <code>pclose()</code> 无法取得子进程的结束状态

2. 标准流管道代码举例

`popen.c` 源代码如下：

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#define BUFSIZE 1024
int main()
{
    FILE *fp;
    char *cmd = "ps -ef";
    char buf[BUFSIZE];
    buf[BUFSIZE] = '\0';
    if((fp=popen(cmd,"r"))==NULL)
        perror("popen");
    while((fgets(buf,BUFSIZE,fp))!=NULL)
        printf("%s",buf);
    pclose(fp);
    exit(0);
}
```

编译 `gcc popen.c -o popen`。

执行 `./popen`，执行结果如下：

```
root      4325      1  0 17:57 tty4      00:00:00 /sbin/getty 38400 tty4
root      4330      1  0 17:57 tty2      00:00:00 /sbin/getty 38400 tty2
.....
```

14.2.3 命名管道（FIFO）

1. 命令管道原理

命名管道（FIFO）和一般的管道相比，有以下一些显著的不同：

- 命名管道是在文件系统中作为一个特殊的设备文件而存在的。
- 不同祖先的进程之间可以通过命名管道共享数据。
- 当共享命名管道的进程执行完所有的 I/O 操作以后，命名管道将继续保存在文件系统中，以便以后使用。
- 普通管道只能由父子兄弟等相关进程使用，它们共同的祖先进程创建了管道。但是，通过命名管道，不相关的进程也能交换数据。
- 一旦用 `mkfifo` 函数创建了一个命名管道，就可用 `open` 打开它。实际上，一般的文件 I/O 函数（`close`、`read`、`write`、`unlink` 等）都可用于命名管道。

2. 命名管道函数原型

命名管道的 `mkfifo` 函数原型及其说明如下。

mkfifo（创建命名管道）	
所需头文件	<code>#include <sys/types.h></code> <code>#include <sys/stat.h></code>
函数说明	<p><code>mkfifo()</code>会依参数 <code>pathname</code> 建立特殊的 FIFO 文件，该文件必须不存在，而参数 <code>mode</code> 为该文件的权限 (<code>mode&~umask</code>)，因此，<code>umask</code> 值也会影响到 FIFO 文件的权限。<code>mkfifo()</code>建立的 FIFO 文件，其他进程都可以用读写一般文件的方式存取。当使用 <code>open()</code>来打开 FIFO 文件时，<code>O_NONBLOCK</code> 旗标会有影响：</p> <p>① 当使用 <code>O_NONBLOCK</code> 旗标时,打开 FIFO 文件来读取的操作会立刻返回，但是若没有其他进程打开 FIFO 文件来读取，则写入的操作会返回 <code>ENXIO</code> 错误代码</p> <p>② 没有使用 <code>O_NONBLOCK</code> 旗标时，打开 FIFO 来读取的操作会等到其他进程打开 FIFO 文件来写入才正常返回。同样，打开 FIFO 文件来写入的操作会等到其他进程打开 FIFO 文件来读取后才正常返回</p>
函数原型	<code>int mkfifo(const char * pathname,mode_t mode)</code>
函数传入值	<code>pathname</code> : 管道文件名
	<code>mode</code> : 管道创建方式
函数返回值	成功: 0
	失败: -1, 错误原因存于 <code>errno</code> 中
错误代码	<p><code>EACCESS</code>: 参数 <code>pathname</code> 所指定的目录路径无可执行的权限</p> <p><code>EEXIST</code>: 参数 <code>pathname</code> 所指定的文件已存在</p> <p><code>ENAMETOOLONG</code>: 参数 <code>pathname</code> 的路径名称太长</p> <p><code>ENOENT</code>: 参数 <code>pathname</code> 包含的目录不存在</p> <p><code>ENOSPC</code>: 文件系统的剩余空间不足</p> <p><code>ENOTDIR</code>: 参数 <code>pathname</code> 路径中的目录存在但却非真正的目录</p> <p><code>EROFS</code>: 参数 <code>pathname</code> 指定的文件存在于只读文件系统内</p> <p><code>ENXIO</code>: 指定的设备或地址不存在</p>

3. 命名管道代码举例

本例展示如何通过命名管道交换数据。

(1) 创建命名管道并写入数据

`fifo_snd.c` 源代码如下：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#define FIFO "/tmp/fifo"
int main()
{
    char buffer[80];
    int fd;
    int n ;
    int ret ;
    char info[80] ;
    unlink(FIFO); /*若存在该管道文件，则删除*/
    ret = mkfifo(FIFO, 0600); /*0600 表明只有该用户进程有读写权限*/
    if ( ret )
    {
```

```

        perror("mkfifo error") ;
        return -1 ;
    }
    memset(info, 0x00, sizeof(info)) ;
    strcpy(info, "happy new year!") ;
    fd = open (FIFO,O_WRONLY);
    n=write(fd, info, strlen(info) );
    if ( n < 0 )
    {
        perror("write error") ;
        return -1 ;
    }
    close(fd);
    return 0 ;
}

```

(2) 从管道中读取数据

fifo_rcv.c 源代码如下:

```

#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#define FIFO  "/tmp/fifo"
int main()
{
    char buffer[80];
    int fd;
    int n ;
    char info[80] ;
    fd= open(FIFO,O_RDONLY);
    n = read(fd,buffer,80);
    if ( n < 0 )
    {
        perror("read error") ;
        return -1 ;
    }
    printf("buffer=%s\n", buffer);
    close(fd);
    return 0 ;
}

```

(3) 编译与执行

- ① 编译 gcc fifo_snd.c -o fifo_snd。
- ② 编译 gcc fifo_rcv.c -o fifo_rcv。
- ③ 在此用户下执行 ./fifo_snd, 可以看见此进程正处于阻塞状态。
- ④ 在此用户其他界面下执行 ./fifo_rcv, 执行结果如下:

```
buffer=happy new year!
```

⑤ 查看此管道文件 `l /tmp/fifo`，此管道文件存在。

```
prw----- 1 zjkf db2iadml 0 2011-01-13 02:31 /tmp/fifo
```

⑥ 在其他用户下执行 `./fifo_rcv`，由于管道文件权限为 0600，执行结果如下：

```
read error: Bad file descriptor
```

14.3 信号

14.3.1 信号概述

信号是进程间通信机制中唯一的异步通信机制，可以看做是异步通知，通知接收信号的进程有哪些事情发生了。

信号同时又是一种软件中断，当某进程接收到信号时，会中止当前程序的执行，去处理信号的注册函数，然后回到断点继续往下执行。

信号事件的发生由两类原因引起，一是由硬件原因引起（按下键盘上的按键或者其他硬件故障），如在终端上按 Delete 键，通常产生中断信号 SIGINT。二是由软件原因引起，如 kill、raise、alarm、setitimer 等系统函数会引起信号的发送，同时除以 0 等非法运算也会引起信号的发送。

进程能对每一个信号设置独立的处理方式：它能忽略该信号，也能设置相应的信号处理程序（称为捕捉），或对信号什么也不做，信号发生的时候执行系统的默认动作。

进程还能通过信号集操作函数设置对某些信号的阻塞（block）标志。如果一个信号被设置为阻塞，当信号发生的时候，它会和正常的信号一样被递送（deliver）给进程，但只有进程解除对该信号的阻塞时才会被处理。也就是说，信号阻塞只阻止信号被处理，但不阻止信号的产生。

从一个信号被递送给进程到该信号得到处理之间的时间间隔称为信号未决（或称信号挂起、信号被搁置）。

所有的信号中，有两个信号（SIGSTOP 和 SIGKILL）是特别的，它们不能被捕捉，也不能被忽略，同时还不能被阻塞，这个特性确保了系统管理员在所有的时间内都能用暂停信号和杀死信号结束某个进程。

1. 信号分类

信号有两种分类方式：从可靠性上，可分为可靠信号和不可靠信号；从与时间的关系上，可分为实时信号和非实时信号。

Linux 信号机制基本上是从 UNIX 系统中继承过来的。早期 UNIX 系统中的信号机制比较简单和原始，后来在实践中暴露出一些问题。因此，把那些建立在早期机制上的信号叫做“不可靠信号”，信号值小于 32 的信号都是不可靠信号，这就是“不可靠信号”的来源。不可靠信号的主要问题是：进程每次处理某个信号后，就对该信号的响应设置为默认动作。在某些情况下，这将

导致对信号的错误处理。因此，用户如果不希望这种结果，那么就要在信号处理函数结尾再一次调用 `signal` 重新安装该信号。其次，不可靠信号还有可能造成信号丢失。因此，早期 UNIX 下的不可靠信号主要指的是进程可能对信号做出错误的反应以及信号的可能丢失。

随着时间的发展，实践证明了有必要对信号的原始机制加以改进和扩充。所以，后来出现的各种 UNIX 版本分别在这方面进行了研究，力图实现“可靠信号”。由于原来定义的信号已有许多应用，不好再做改动，最终只好又新增加了一些信号，并在一开始就把它们定义为可靠信号，这些信号支持排队，不会丢失。同时，信号的发送和安装也出现了新版本的信号发送函数 `sigqueue()` 和信号安装函数 `sigaction()`。POSIX 4 对可靠信号机制做了标准化处理。但是，POSIX 只对可靠信号机制应具有的功能以及信号机制的对外接口做了标准化处理，对信号机制的实现没有做具体的规定。信号值位于 32 和 64 之间的信号都是可靠信号，可靠信号克服了信号可能丢失的问题。

Linux 支持不可靠信号，但是对不可靠信号机制做了改进：在调用完信号处理函数后，不必重新调用该信号的安装函数，Linux 信号安装函数 `sigaction` 是在可靠机制上实现的。因此，Linux 下不可靠信号的问题主要指的是信号可能丢失。

Linux 在支持新版本的信号安装函数 (`sigaction`) 以及信号发送函数 (`sigqueue`) 的同时，仍然支持早期的信号安装函数 (`signal`) 和信号发送函数 (`kill`)。

不要认为：由 `sigqueue` 发送、由 `sigaction` 安装的信号就是可靠的。事实上，可靠信号是指后来添加的新信号（信号值位于 32 和 64 之间），不可靠信号是指信号值小于 32 的信号。信号的可靠与不可靠只与信号值有关，与信号的发送和安装函数无关。目前，Linux 系统中的 `signal` 是通过 `sigaction` 函数封装实现的。因此，即使通过 `signal` 安装的信号，在信号处理函数的结尾也不必再调用一次信号安装函数。同时，由 `signal` 安装的实时信号支持排队，同样不会丢失。

对于目前 Linux 的两个信号安装函数 `signal` 和 `sigaction` 来说，它们都不能把 32 以前的信号变成可靠信号（1~31 的信号不支持排队，仍有可能丢失，仍然是不可靠信号），而对 32 以后的信号都支持排队。这两个函数的最大区别在于：经过 `sigaction` 安装的信号都能传递信息给信号处理函数，而经过 `signal` 安装的信号却不能向信号处理函数传递信息。

非实时信号（1~31）都不支持排队，都是不可靠信号；实时信号（32~64）都支持排队，都是可靠信号。实时信号之所以是可靠的，是因为在该进程阻塞等待处理的时间内，发送给该进程的所有实时信号会排队等待；而对于非实时信号，则系统只会处理第一个信号，在该进程阻塞的时间内，后续信号被简单丢弃。早期的 `kill` 函数只能向特定的进程发送一个特定的信号，并且早期的信号处理函数也不能接收附加数据，而 `sigqueue` 和 `sigaction` 解决了这个问题。

`kill -l` 命令可用于查看信号列表。信号可以由数字表示，也可以用 SIG 开头的宏表示。信号值为 1~31 的信号是传统 UNIX 支持的信号，是不可靠信号（非实时信号）；信号值为 32~63 的信号是后来扩充的，称为可靠信号（实时信号）。现在在 Linux 系统中，不可靠信号和可靠信号的区别在于前者不支持排队，可能会造成信号的丢失，而后者不会。

2. 信号源分类

内核为进程产生信号来说明不同的事件，这些事件就是信号源。主要的信号源有如下七种。

- ① 异常：进程运行过程中出现异常。
- ② 其他进程：一个进程可以向另外一个或一组进程发送信号。
- ③ 终端中断：Ctrl-C、Ctrl-\等。
- ④ 作业控制：前台、后台进程的管理。
- ⑤ 分配额：CPU 超时或文件大小突破限制。
- ⑥ 通知：通知进程某事件发生，如 I/O 就绪等。
- ⑦ 报警：计时器到期。

3. 不可靠信号说明

表 14-1 列出了不可靠信号名称及其说明。在实际应用的编程中，用到的主要还是不可靠信号（即信号值为 1~31 的信号）。

表 14-1 不可靠信号列表

信号值	信号宏名称	默认动作	说 明
1	SIGHUP	终止进程	从终端上发出的结束信号
2	SIGINT	终止进程	中断进程
3	SIGQUIT	建立 CORE 文件	终止进程，并且生成 core 文件
4	SIGILL	建立 CORE 文件	非法指令
5	SIGTRAP	建立 CORE 文件	跟踪自陷
6	SIGABRT	建立 CORE 文件	异常终止
7	SIGBUS	建立 CORE 文件	总线错误
8	SIGFPE	建立 CORE 文件	浮点异常
9	SIGKILL	终止进程	杀死进程
10	SIGUSR1	终止进程	用户定义信号 1
11	SIGSEGV	建立 CORE 文件	段非法错误
12	SIGUSR2	终止进程	用户定义信号 2
13	SIGPIPE	终止进程	向一个没有读进程的管道写数据
14	SIGALARM	终止进程	计时器到时
15	SIGTERM	终止进程	软件终止信号
16	SIGSTKFLT	终止进程	Linux 专用，数学协处理器的栈异常
17	SIGCHLD	忽略信号	当子进程停止或退出时通知父进程
18	SIGCONT	忽略信号	继续执行一个停止的进程
19	SIGSTOP	停止进程	非终端来的停止信号
20	SIGTSTP	停止进程	终端来的停止信号
21	SIGTTIN	停止进程	后台进程读终端

续表

信号值	信号宏名称	默认动作	说 明
22	SIGTTOU	停止进程	后台进程写终端
23	SIGURG	忽略信号	I/O 紧急信号
24	SIGXGPU	终止进程	超过 CPU 时间限制
25	SIGXFSZ	终止进程	进程超过文件长度限制
26	SIGVTALRM	终止进程	虚拟计时器超时
27	SIGPROF	终止进程	统计分布计时器超时
28	SIGWINCH	忽略信号	窗口大小发生变化
29	SIGIO	忽略信号	异步 I/O 事件
30	SIGPWR	忽略信号	电源失效再启动
31	SIGSYS	建立 CORE 文件	非法系统调用

4. 常见信号说明

表 14-2 列出了常见的信号及其说明。

表 14-2 常见信号列表

信号宏名称	信号说明
SIGHUP	从终端上发出的结束信号
SIGINT	来自键盘的中断信号（Ctrl-C）
SIGQUIT	来自键盘的退出信号（Ctrl-\）
SIGFPE	浮点异常信号（例如浮点运算溢出）
SIGKILL	该信号结束接收信号的进程
SIGALRM	进程的定时器到期时，发送该信号
SIGTERM	kill 命令发出的信号
SIGCHLD	标识子进程停止或结束的信号
SIGSTOP	来自键盘（Ctrl-Z）或调试程序的停止执行信号

5. 信号的三种操作方式

信号具有以下三种操作方式。

- ① 忽略此信号，SIG_IGN 常数表示信号函数的忽略。但 SIGKILL 和 SIGSTOP 信号不能忽略。
- ② 捕捉信号。在某种信号发生时，调用一个用户自定义函数，在用户自定义函数中可执行用户希望对这个事件进行的处理。
- ③ 执行系统的默认动作，SIG_DFL 常数表示信号函数的默认值。对大多数信号来说，系统的默认动作是终止该进程。

6. 信号的五种默认动作

信号发生时，对接收信号进程的处理有如下五种默认动作。

- ① 异常终止 (abort)：在进程的当前目录下，把进程的地址空间内容、寄存器内容保存到一个叫做 core 的文件中，而后终止该进程。
- ② 退出 (exit)：不产生 core 文件，直接终止该进程。
- ③ 忽略 (ignore)：忽略该信号。
- ④ 停止 (stop)：挂起该进程。
- ⑤ 继续 (continue)：如果进程被挂起，则恢复进程的运行。否则，忽略信号。

7. 阻塞信号和忽略信号两种操作的区别

阻塞信号允许信号被发送给进程，但不进行处理，需要等到阻塞解除后再处理。而忽略信号是进程根本不接收该信号，所有被忽略的信号都被简单丢弃。

用系统调用 sigprocmask 可设置信号是否被阻塞，而系统调用 sigaction 和库函数 signal 则设置进程是否忽略一个信号。

14.3.2 信号的发送和捕捉函数

下面是信号中常用到的信号发送和捕捉函数，其中 alarm 和 kill 两个信号发送函数在应用编程中最常用。

1. alarm 函数

(1) alarm 函数原型

alarm（设置信号传送闹钟）	
所需头文件	#include <unistd.h> #include <signal.h>
函数说明	alarm() 用来设置信号 SIGALRM 在经过参数 seconds 指定的秒数后传送给目前的进程。如果参数 seconds 为 0，则之前设置的闹钟会被取消，并将剩下的时间返回
函数原型	unsigned int alarm(unsigned int seconds)
函数返回值	返回之前闹钟的剩余秒数，如果之前未设闹钟，则返回 0

(2) alarm 函数说明

当所设置的时间值超过后，产生 SIGALRM 信号。如果不忽略或不捕捉此信号，则其默认动作是终止该进程。

每个进程只能有一个闹钟时间。如果在调用 alarm 时，以前已为该进程设置过闹钟时间，而且它还没有超时，则该闹钟时间的剩余时间值作为本次 alarm 函数调用的值返回，以前登记的闹钟时间则被新值代换。

如果有以前登记的尚未超过的闹钟时间，而新设的闹钟时间值为 0，则取消以前的闹钟时间，其剩余时间值仍作为函数的返回值。

(3) alarm 函数举例

alarm.c 源代码如下:

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
void handler() { /*信号处理函数*/
    printf("hello\n");
}
int main()
{
    int i;
    int time ;
    signal(SIGALRM,handler); /*注册 SIGALRM 信号处理方式*/
    alarm(3);
    for(i=1;i<5;i++){
        printf("sleep %d ...\n", i);
        sleep(1);
    }

    alarm(3);
    sleep(2);
    time=alarm(0) ; /*取消 SIGALRM 信号, 返回剩余秒数*/
    printf("time=%d\n",time) ;
    for(i=1;i<3;i++){
        printf("sleep %d ...\n", i);
        sleep(1);
    }

    return 0 ;
}
```

编译 gcc alarm.c -o alarm。

执行 ./alarm, 执行结果如下:

```
sleep 1 ...
sleep 2 ...
sleep 3 ...
hello
sleep 4 ...
time=1
sleep 1 ...
sleep 2 ...
```

对程序结果分析如下:

信号是一种软中断, 中断的原理是保留执行现场, 跳转到中断函数处执行, 执行完毕恢复以前的现场, 在断点处继续往下执行。所以, alarm 函数经过 3 秒时, for 循环已经到了 sleep 的 3 秒位置; alarm 发送闹铃信号, 引起中断, 程序执行 SIGALRM 注册函数 handler (中断处理函数), 即输出 hello。信号处理 handler 函数执行完后, 应用程序从中断点恢复, 继续执行 for 循环, 此时 sleep 到了 4 秒的位置。

alarm(0)取消注册的闹铃（SIGALRM）信号，所以，第二次没有执行 SIGALRM 信号注册函数。

2. kill 函数

(1) kill 函数原型

kill 函数是将信号发送给指定的 pid 进程。普通用户利用 kill 函数将信号发送给该用户下任意一个进程，而特权用户（root）可以将信号发送给系统中的任意一个进程。

kill 函数原型及说明如下。

kill（传送信号给指定的进程）		
所需头文件	#include <sys/types.h> #include <signal.h>	
函数说明	kill() 可以用来传送参数 sig 指定的信号给参数 pid 指定的进程	
函数原型	int kill(pid_t pid,int sig)	
函数传入值	pid	pid>0 将信号传给进程识别码为 pid 的进程
		pid=0 将信号传给和目前进程相同进程组的所有进程
		pid=-1 将信号广播传送给系统内所有的进程
		pid<0 将信号传给进程组识别码为 pid 绝对值的所有进程
	sig	信号编号
函数返回值	成功: 0	
	出错: -1, 错误原因存于 error 中	
错误代码	EINVAL: 参数 sig 不合法 ESRCH: 参数 pid 所指定的进程或进程组不存在 EPERM: 权限不够无法传送信号给指定进程	

(2) kill 函数举例

下列代码实现的是父进程发信号给子进程。

kill.c 源代码如下：

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
int main()
{
    pid_t pid;
    int status;

    pid= fork() ;
    if( 0==pid ){
        printf("Hi I am child process!\n");
        sleep(10);
    }
```

```
else if ( pid > 0 ){
    printf("send signal to child process (%d) \n",pid);
    sleep(1);
    /*发送 SIGABRT 信号给子进程，此信号引起接收进程异常终止*/
    kill(pid ,SIGABRT);
    /*等待子进程返回终止信息*/
    wait(&status);
    if(WIFSIGNALED(status))
        printf("chile process receive signal %d\n",WTERMSIG(status));
}else{
    perror("fork error") ;
    return -1 ;
}

return 0 ;
}
```

编译 gcc kill.c -o kill。

执行./kill，执行结果如下：

```
Hi I am child process!
send signal to child process (10498)
chile process receive signal 6
```

3. raise 函数

与 kill 函数不同的是，raise 函数运行是向进程自身发送信号。

raise 函数的原型及其说明如下。

raise（向自己发送信号）	
所需头文件	#include <sys/types.h> #include <signal.h>
函数说明	向自己发送信号
函数原型	int raise(int sig)
函数传入值	sig: 信号
函数返回值	成功: 0
	出错: -1, 错误原因存于 error 中

4. pause 函数

(1) pause 函数原型

pause（让进程暂停直到信号出现）	
所需头文件	#include <unistd.h>
函数说明	令目前的进程暂停（进入睡眠状态），直到被信号（signal）中断
函数原型	int pause(void)
函数返回值	只返回-1，错误原因存于 error 中
错误代码	EINTR: 有信号到达并中断了此函数

(2) pause 函数举例

下面的 pause.c 源代码简单地实现了 sleep 函数的功能。由于 SIGALRM 信号默认的系统动作为终止进程，所以，在程序调用 pause 之后就暂停，当 3 秒钟后接收到 alarm 信号时，进程就终止。

pause.c 源代码如下：

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int ret ;
    ret=alarm(3) ; /*调用 alarm 定时器函数*/
    pause() ;
    printf("I have been waken up.\n") ;
    return 0 ;
}
```

编译 gcc pause.c -o pause。

执行 ./pause，执行结果如下：

Alarm clock

5. sleep 和 abort

(1) sleep 函数原型

sleep（让进程暂停执行一段时间）	
所需头文件	#include <unistd.h>
函数说明	sleep() 会令目前的进程暂停，直到达到参数 seconds 所指定的时间，或是被信号中断
函数原型	unsigned int sleep(unsigned int seconds)
函数传入值	seconds: 睡眠的秒数
函数返回值	若进程在参数 seconds 所指定的时间暂停，则返回 0；若有信号中断，则返回剩余秒数

(2) abort 函数原型

abort（以异常方式结束进程）	
所需头文件	#include <stdlib.h>
函数说明	此函数将 SIGABRT 信号给调用进程，此信号将引起调用进程的异常终止。此时所有已打开的文件流会自动关闭，所有的缓冲区数据会自动写回
函数原型	void abort(void)

(3) sleep、abort 函数举例

sys_sleep.c 源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(void)
{
    system("pwd") ;
    sleep(9) ; /*wait 9 second*/
    printf("Calling abort()\n");
    abort();
    printf("abort() after\n") ;
    return 0; /* This is never reached */
}
```

编译 `gcc sys_sleep.c -o sys_sleep`。

执行 `./sys_sleep`，执行结果如下：

```
/home/zjzf/public/signal
Calling abort()
```

6. 信号的发送与捕捉简要总结

对上述信号的发生与捕捉函数简要总结说明如下：

- `kill` 函数可以向有用户权限的任何进程发送信号，通常用 `kill` 函数来结束进程。
- 与 `kill` 函数不同的是，`raise` 函数只向进程自身发送信号。
- 使用 `alarm` 函数可以设置一个时间值（闹铃时间），在将来的某个时刻，该时间值超过时发送信号。
- `pause` 函数使调用进程挂起，直至捕捉到一个信号。

14.3.3 信号的处理

信号是与一定的进程相联系的。也就是说，一个进程可以决定在进程中对哪些信号进行什么样的处理。例如，一个进程可以忽略某些信号而只处理其他一些信号。另外，一个进程还可以选择如何处理信号。总之，这些总与特定的进程相联系。因此，首先要建立其信号和进程的对应关系，这就是信号的安装登记。

Linux 主要有两个函数实现信号的安装登记：`signal` 和 `sigaction`。其中 `signal` 在系统调用的基础上实现，是库函数，它只有两个参数，不支持信号传递信息，主要用于前 32 个非实时信号的安装。`sigaction` 是较新的函数（由 `sys_signal` 和 `sys_rt_sigaction` 两个系统调用实现），有三个参数，支持信号传递信息，主要用来与 `sigqueue` 系统调用配合使用。当然，`sigaction` 同样支持非实时信号的安装，`sigaction` 优于 `signal` 主要体现在支持信号带有参数。

对于应用程序自行处理的信号来说，信号的生命周期要经过信号的安装登记、信号集操作、信号的发送和信号的处理四个阶段。信号的安装登记指的是在应用程序中，安装对此信号的处理方法。信号集操作的作用是用于对指定的一个或多个信号进行信号屏蔽，此阶段对有些应用程序来说并不需要。信号的发送指的是发送信号，可以通过硬件（如在终端上按下 `Ctrl+C` 组合键）发送的信号和软件（如通过 `kill` 函数）发送的信号。信号的处理指的是操作系统对接收信号进

程的处理，处理方法是先检查信号集操作函数是否对此信号进行屏蔽，如果没有屏蔽，操作系统将按信号安装函数中登记注册的处理函数完成对此进程的处理。

1. signal 函数

(1) 函数说明

在 signal 函数中有两个形参，分别代表需要处理的信号编号值和处理信号函数的指针。它主要用于前 32 种非实时信号的处理，不支持信号的传递信息。但是由于使用简单，易于理解，因此，在许多场合被程序员使用。

对于 UNIX 系统来说，使用 signal 函数时，自定义处理信号函数执行一次后失效，对该信号的处理回到默认处理方式。下面以一个例子进行说明，例如，在一个程序中使用 signal(SIGQUIT,my_func)函数调用，其中 my_func 是自定义函数。应用进程收到 SIGQUIT 信号时，会跳转到自定义处理信号函数 my_func 处执行，执行后，信号注册函数 my_func 失效，对 SIGQUIT 信号的处理回到操作系统的默认处理方式，当应用进程再次收到 SIGQUIT 信号时，会按操作系统默认的处理方式进行处理（即不再执行 my_func 处理函数）。而在 Linux 系统中，signal 函数已被改写，由 sigaction 函数封装实现，则不存在上述问题。

(2) signal 函数原型

signal（设置信号处理方式）		
所需头文件	#include <signal.h>	
函数说明	设置信号处理方式。signal()会依参数 signum 指定的信号编号来设置该信号的处理函数。当指定的信号到达时，就会跳转到参数 handler 指定的函数执行	
函数原型	void (*signal(int signum,void(* handler)(int)))(int)	
函数传入值	signum	指定信号编号
	handle	SIG_IGN: 忽略参数 signum 指定的信号
		SIG_DFL: 将参数 signum 指定的信号重设为核心预设的信号处理方式，即采用系统默认方式处理信号
		自定义信号函数处理指针
函数返回值	成功	返回先前的信号处理函数指针
	出错	SIG_ERR(-1)
附加说明	在 UNIX 环境中，在信号发生跳转到自定义的 handler 处理函数执行后，系统会自动将此处理函数换回原来系统预设的处理方式，如果要改变此情形，则要用 sigaction 函数。在 Linux 环境中不存在此问题	

signal 函数原型比较复杂，如果使用下面的 typedef，则可使其简化。

```
typedef void sign(int);
sign *signal(int, handler *);
```

可见，该函数原型首先整体指向一个无返回值带一个整型参数的函数指针，也就是信号的原始配置函数。接着该原型又带有两个参数，其中的第二个参数可以是用户自定义的信号处理函数的函数指针。对这个函数格式可以不理解，但需要学会使用。

(3) signal 函数使用示例

该示例表明了如何使用 signal 函数进行安装登记信号处理函数。当该信号发生时，登记的信号处理函数会捕捉到相应的信号，并做出相应的处理。这里，my_func 就是信号处理的函数指针。读者还可以将 my_func 改为 SIG_IGN 或 SIG_DFL 查看运行结果。

signal.c 源代码如下：

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
/*自定义信号处理函数*/
void my_func(int sign_no)
{
    if(sign_no==SIGINT)
        printf("I have get SIGINT\n");
    else if(sign_no==SIGQUIT)
        printf("I have get SIGQUIT\n");
}
int main()
{
    printf("Waiting for signal SIGINT or SIGQUIT \n ");
    /*发出相应的信号，并跳转到信号处理函数处*/
    signal(SIGINT, my_func);
    signal(SIGQUIT, my_func);
    pause();
pause();
    exit(0);
}
```

编译 gcc signal.c -o signal。

执行 ./signal，执行结果如下：

```
Waiting for signal SIGINT or SIGQUIT
I have get SIGINT      /*按下 Ctrl+C 组合键，操作系统就会向进程发送 SIGINT 信号*/
I have get SIGQUIT     /*按下 Ctrl+\组合键（退出），操作系统就会向进程发送 SIGQUIT 信号*/
```

2. sigaction 函数

(1) sigaction 函数原型

sigaction 函数用来查询和设置信号处理方式，它用来替换早期的 signal 函数。sigaction 函数原型及说明如下。

sigaction（查询和设置信号处理方式）	
所需头文件	#include <signal.h>
函数说明	sigaction()会依参数 signalnum 指定的信号编号来设置该信号的处理函数
函数原型	int sigaction(int signalnum,const struct sigaction *act ,struct sigaction *oldact)

续表

sigaction (查询和设置信号处理方式)		
函数传入值	signum	可以指定 SIGKILL 和 SIGSTOP 以外的所有信号
	act	<p>参数结构 sigaction 定义如下</p> <pre>struct sigaction { void (*sa_handler) (int); void (*sa_sigaction)(int, siginfo_t *, void *); sigset_t sa_mask; int sa_flags; void (*sa_restorer) (void); }</pre> <p>① sa_handler: 此参数和 signal() 的参数 handler 相同, 此参数主要用来对信号旧的安装函数 signal() 处理形式的支持</p> <p>② sa_sigaction: 新的信号安装机制, 处理函数被调用的时候, 不但可以得到信号编号, 而且可以获悉被调用的原因以及产生问题的上下文的相关信息</p> <p>③ sa_mask: 用来设置在处理该信号时暂时将 sa_mask 指定的信号搁置</p> <p>④ sa_restorer: 此参数没有使用</p> <p>⑤ sa_flags: 用来设置信号处理的其他相关操作, 有下列数值可用 (可与 OR 运算 () 组合)</p> <ul style="list-style-type: none">■ A_NOCLDSTOP: 如果参数 signum 为 SIGCHLD, 则当子进程暂停时并不会通知父进程■ SA_ONESHOT/SA_RESETHAND: 当调用新的信号处理函数前, 将此信号处理方式改为系统预设的方式■ SA_RESTART: 被信号中断的系统调用会自行重启■ SA_NOMASK/SA_NODEFER: 在处理此信号未结束前, 不理睬此信号的再次到来■ SA_SIGINFO: 信号处理函数是带有三个参数的 sa_sigaction
	oldact	如果参数 oldact 不是 NULL 指针, 则原来的信号处理方式会由此结构 sigaction 返回
函数返回值	成功: 0	
	出错: -1, 错误原因存于 error 中	
附加说明	<p>信号处理安装的新旧两种机制。</p> <p>① 使用旧的处理机制: struct sigaction act; act.sa_handler=handler_old;</p> <p>② 使用新的处理机制: struct sigaction act; act.sa_sigaction=handler_new;</p> <p>并设置 sa_flags 的 SA_SIGINFO 位</p>	
错误代码	<p>EINVAL: 参数 signum 不合法, 或是企图拦截 SIGKILL/SIGSTOP 信号</p> <p>EFAULT: 参数 act、oldact 指针地址无法存取</p> <p>EINTR: 此调用被中断</p>	

(2) sigaction 函数使用示例

sigaction.c 源代码如下:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/types.h>
#include <unistd.h>
void new_op(int, siginfo_t *, void *);
int main(int argc, char**argv)
```

```

{
    struct sigaction act;
    int sig;
    sig=atoi(argv[1]);
    sigemptyset(&act.sa_mask);
    act.sa_flags=SA_SIGINFO;
    act.sa_sigaction=new_op;
    if(sigaction(sig,&act,NULL) < 0)
    {
        perror("install signal error");
        return -1 ;
    }
    while(1)
    {
        sleep(2);
        printf("wait for the signal\n");
    }

    return 0 ;
}
void new_op(int signum,siginfo_t *info,void *myact)
{
    printf("receive signal %d\n", signum);
    sleep(5);
}

```

编译 gcc sigaction.c -o sigaction。

执行 ./sigaction 2, 执行结果如下:

```

wait for the signal
receive signal 2      /*按下 Ctrl+C */
退出                 /*按下 Ctrl-\ */

```

3. 信号集操作函数

由于有时需要把多个信号当做一个集合进行处理, 这样信号集就产生了。信号集用来描述一类信号的集合, Linux 所支持的信号可以全部或部分出现在信号集中。信号集操作函数最常用于信号屏蔽。比如, 有时候希望某个进程正确执行, 而不想进程受到一些信号的影响, 此时就需要用到信号集操作函数完成对这些信号的屏蔽。

信号集操作函数按照功能和使用顺序分为三类, 分别为创建信号集函数、设置信号屏蔽位函数和查询被搁置(未决)的信号函数。创建信号集函数只是创建一个信号的集合, 设置信号屏蔽位函数对指定信号集中的信号进行屏蔽, 查询被搁置的信号函数用来查询当前“未决”的信号集。信号集函数组并不能完成信号的安装登记工作, 信号的安装登记需要通过 sigaction 函数或 signal 函数来完成。

查询被搁置的信号是信号处理的后续步骤, 但不是必需的。由于有时进程在某时间段内要求阻塞一些信号, 程序完成特定工作后解除对该信号的阻塞, 这个时间段内被阻塞的信号称为“未决”信号。这些信号已经产生, 但没有被处理, sigpending 函数用来检测进程的这些“未决”信号, 并进一步决定对它们做何种处理(包括不处理)。

(1) 创建信号集函数

创建信号集函数有如下五个。

- ① sigemptyset：初始化信号集合为空。
- ② sigfillset：把所有的信号加入到集合中，信号集中将包含 Linux 支持的 64 种信号。
- ③ sigaddset：将指定的信号加入到信号集合中。
- ④ sigdelset：将指定的信号从信号集中删去。
- ⑤ sigismember：查询指定的信号是否在信号集合中。

创建信号集合函数原型及说明如下。

创建信号集合函数原型	
所需头文件	#include <signal.h>
函数原型	int sigemptyset(sigset_t *set)
	int sigfillset(sigset_t *set)
	int sigaddset(sigset_t *set,int signum)
	int sigdelset(sigset_t *set,int signum)
	int sigismember(sigset_t *set,int signum)
函数传入值	set: 信号集
	signum: 指定信号值
函数返回值	成功: 0 (sigismember 函数例外, 成功返回 1, 失败返回 0)
	出错: -1, 错误原因存于 error 中

(2) 设置信号屏蔽位函数

每个进程都有一个用来描述哪些信号递送到进程时将被阻塞的信号集，该信号集中的所有信号在递送到进程后都将被阻塞。调用函数 sigprocmask 可设定信号集内的信号阻塞或不阻塞。其函数原型及说明如下。

sigprocmask (设置信号屏蔽位)		
所需头文件	#include <signal.h>	
函数原型	int sigprocmask(int how,const sigset_t *set,sigset_t *oset)	
函数传入值	how (决定函数的操作方式)	SIG_BLOCK: 增加一个信号集合到当前进程的阻塞集合中
		SIG_UNBLOCK: 从当前的阻塞集合中删除一个信号集合
		SIG_SETMASK: 将当前的信号集合设置为信号阻塞集合
	set: 指定信号集	
	oset: 信号屏蔽字	
函数返回值	成功: 0	
	出错: -1, 错误原因存于 error 中	

(3) 查询被搁置（未决）信号函数

sigpending 函数用来查询“未决”信号，其函数原型及说明如下。

sigpending（查询未决信号）	
所需头文件	#include <signal.h>
函数说明	将被搁置的信号集合由参数 set 指针返回
函数原型	int sigpending(sigset_t *set)
函数传入值	set: 要检测信号集
函数返回值	成功: 0
	出错: -1, 错误原因存于 error 中
错误代码	EFAULT: 参数 set 指针地址无法存取
	EINTR: 此调用被中断

（4）对信号集操作函数的使用方法

对信号集操作函数的使用方法和顺序如下：

- ① 使用 signal 或 sigaction 函数安装和登记信号的处理。
- ② 使用 sigemptyset 等定义信号集函数完成对信号集的定义。
- ③ 使用 sigprocmask 函数设置信号屏蔽位。
- ④ 使用 sigpending 函数检测未决信号，非必需步骤。

（5）信号集操作函数使用实例

该实例首先使用 sigaction 函数对 SIGINT 信号进行安装登记，安装登记使用了新旧两种机制，其中用 #if 0 注释掉的部分为信号安装的新机制。接着程序把 SIGQUIT、SIGINT 两个信号加入信号集，并把该信号集设为阻塞状态。程序开始睡眠 30 秒，此时用户按下 Ctrl+C 组合键，程序将测试到此未决信号（SIGINT）；随后程序再睡眠 30 秒后对 SIGINT 信号解除阻塞，此时将处理 SIGINT 登记的信号函数 my_func。最后可以用 SIGQUIT（按 Ctrl+\组合键）信号结束进程执行。

sigset.c 源代码如下：

```
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
/*自定义的信号处理函数*/
#if 0
void my_funcnew(int signum, siginfo_t *info,void *myact)
#endif
void my_func(int signum)
{
    printf("If you want to quit,please try SIGQUIT\n");
}
int main()
{
```

```
sigset_t set, pendset;
struct sigaction action1, action2;

/*设置信号处理方式*/
sigemptyset(&action1.sa_mask);

#if 0 /*信号新的安装机制*/
    action1.sa_flags= SA_SIGINFO;
    action1.sa_sigaction=my_funcnew;
#endif
/*信号旧的安装机制*/
action1.sa_flags= 0;
action1.sa_handler=my_func;
sigaction(SIGINT, &action1, NULL);

/*初始化信号集为空*/
if(sigemptyset(&set)<0)
{
    perror("sigemptyset");
    return -1 ;
}
/*将相应的信号加入信号集*/
if(sigaddset(&set, SIGQUIT)<0)
{
    perror("sigaddset");
    return -1 ;
}
if(sigaddset(&set, SIGINT)<0)
{
    perror("sigaddset");
    return -1 ;
}

/*设置信号集屏蔽字*/
if(sigprocmask(SIG_BLOCK, &set, NULL)<0)
{
    perror("sigprocmask");
    return -1 ;
}
else
{
    printf("blocked\n");
}

/*测试信号是否加入该信号集*/
if(sigismember(&set, SIGINT)){
    printf("SIGINT in set\n") ;
}

sleep( 30 ) ;
/*测试未决信号*/
if ( sigpending(&pendset) <0 )
{
    perror("get pending mask error");
}
```

```
if(sigismember(&pendset, SIGINT) )
{
    printf("signal SIGINT is pending\n");
}

sleep(30) ;
if(sigprocmask(SIG_UNBLOCK,&set,NULL)<0)
{
    perror("sigprocmask");
    return -1 ;
}
else
    printf("unblock\n");

while(1)
{
    sleep(1) ;
}

return 0 ;
}
```

编译 gcc sigset.c -o sigset。

执行 ./sigset, 执行结果如下:

```
blocked
SIGINT in set  /*按下 Ctrl+C */
signal SIGINT is pending
If you want to quit,please try SIGQUIT /*按下 Ctrl+C */
退出
```

第 15 章

System V 进程间通信

System V 曾经也被称为 AT&T System V，是UNIX操作系统众多版本中的一个。它最初由 AT&T 开发，在 1983 年第一次发布，一共发行了 4 个 System V 的主要版本：版本 1、版本 2、版本 3 和版本 4。System V Release 4（简称为 SVR4）是最成功的版本，并成为 UNIX 一些共同特性的源头。System V 是 AT&T 的第一个商业 UNIX 版本（UNIX System III）的加强。System V IPC（Inter-Process Communication，进程间通信）包括三种进程通信方式，即消息队列、信号量和共享内存。这三种方式完全被 Linux 系统继承和兼容。

15.1 System V 进程间通信的键值

消息队列、信号量、共享内存常用在 Linux 服务端编程的进程间通信环境中，这三类编程函数在实际项目中都是用 System V IPC 函数实现的。System V IPC 函数名称和说明如表 15-1 所示。

表 15-1 System V IPC 函数

函数名称	消息队列	信号量	共享内存区
头文件	<sys/msg.h>	<sys/sem.h>	<sys/shm.h>
创建或打开 IPC 函数	msgget	semget	shmget
控制 IPC 操作的函数	msgctl	semctl	shmctl
IPC 操作函数	msgsnd msgrcv	semop	shmat shmdt

1. key_t 键和 ftok 函数

函数 ftok 把一个已存在的路径名和一个整数标识符转换成一个 key_t 值，称为 IPC 键值（也称为 IPC key 键值）。ftok 函数原型及说明如下。

ftok（把一个已存在的路径名和一个整数标识符转换成 IPC 键值）	
所需头文件	#include <sys/types.h> #include <sys/ipc.h>

续表

ftok (把一个已存在的路径名和一个整数标识符转换成 IPC 键值)	
函数说明	把从 pathname 导出的信息与 proj_id 二进制的低序 8 位组合成一个整数 IPC 键
函数原型	key_t ftok(const char *pathname, int proj_id)
函数传入值	pathname: 指定的文件, 此文件必须存在且可存取
	proj_id: 计划代号 (project ID)
函数返回值	成功: 返回 key_t 值 (即 IPC 键值)
	出错: -1, 错误原因存于 error 中
附加说明	key_t 一般为 32 位的 int 型的重定义

ftok 的典型实现是调用 stat 函数, 然后组合以下三个值:

- ① pathname 所在的文件系统的信息 (stat 结构的 st_dev 成员)。
- ② 该文件在本文件系统内的索引节点号 (stat 结构的 st_ino 成员)。
- ③ proj_id 二进制的低序 8 位 (不能为 0)。

上述三个值组合产生一个 32 位键。

2. ftok 函数代码举例

ftok.c 源代码如下:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/stat.h>
#include <unistd.h>
int main(int argc, char **argv)
{
    struct stat stat1 ;
    if ( argc != 2 )
    {
        printf("usage: ftok < pathname >" );
        exit(1) ;
    }
    stat( argv[1], &stat1 ) ;
    printf("st_dev:%lx, st_ino:%lx, key:%x\n", \
(unsigned long)stat1.st_dev, (unsigned long)stat1.st_ino , ftok(argv[1],0x579 )) ;
    printf("st_dev:%lx, st_ino:%lx, key:%x\n", \
(unsigned long)stat1.st_dev, (unsigned long)stat1.st_ino , ftok(argv[1],0x118 )) ;
    printf("st_dev:%lx, st_ino:%lx, key:%x\n", \
(unsigned long)stat1.st_dev, (unsigned long)stat1.st_ino , ftok(argv[1],0x22 )) ;
    printf("st_dev:%lx, st_ino:%lx, key:%x\n", \
(unsigned long)stat1.st_dev, (unsigned long)stat1.st_ino , ftok(argv[1],0x33 )) ;
    exit(0) ;
}
```

编译 gcc ftok.c -o ftok

运行 ./ftok /tmp, 执行结果如下:

```
st_dev:801, st_ino:4db21, key:7901db21
st_dev:801, st_ino:4db21, key:1801db21
st_dev:801, st_ino:4db21, key:2201db21
st_dev:801, st_ino:4db21, key:3301db21
st_dev:801, st_ino:4db21, key:4401db21
```

从上面的程序可以看出，通过 `ftok` 返回的是根据文件（`pathname`）信息和计划编号（`proj_id`）合成的 IPC `key` 键值，从而避免了用户使用 `key` 值的冲突。`proj_id` 值的意义是使一个文件也能生成多个 IPC `key` 键值。`ftok` 利用同一文件最多可得到 IPC `key` 键值 `0xff`（即 256）个，因为 `ftok` 只取 `proj_id` 值二进制的后 8 位，即十六进制数的后两位与文件信息合成 IPC `key` 键值。

3. IPC 键值与 IPC 标识符

- (1) `key` 值选择方式
- 对于 `key` 值，应用程序有如下三种选择：
- ① 调用 `ftok`，给它传递 `pathname` 和 `proj_id`，操作系统根据两者合成 `key` 值。
- ② 指定 `key` 为 `IPC_PRIVATE`，内核保证创建一个新的、唯一的 IPC 对象，IPC 标识符与内存中的标识符不会冲突。`IPC_PRIVATE` 为宏定义，其值等于 0。
- ③ 指定 `key` 为大于 0 的常数，这需要用户自行保证生成的 IPC `key` 值不与系统中存在的冲突，而前两种是操作系统保证的。

(2) IPC 标识符

给 `semget`、`msgget`、`shmget` 传入 `key` 值，它们返回的都是相应的 IPC 对象标识符。注意，IPC 键值和 IPC 标识符是两个概念，后者是建立在前者之上的。图 15-1 画出了从 IPC 键值生成 IPC 标识符图，其中 `key` 为 IPC 键值，由 `ftok` 函数生成；`ipc_id` 为 IPC 标识符，由 `semget`、`msgget`、`shmget` 函数生成。`ipc_id` 在信号量函数中称为 `semid`，在消息队列函数中称为 `msgid`，在共享内存函数中称为 `shmid`，它们表示的是各自的 IPC 对象标识符。

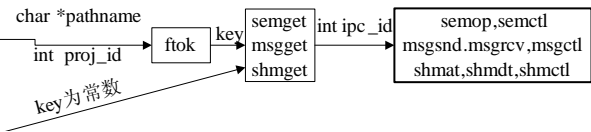


图 15-1 从 IPC 键值生成 IPC 标识符图

4. ipc_perm 结构说明

系统为每一个 IPC 对象保存一个 `ipc_perm` 结构体，该结构说明了 IPC 对象的权限和所有者，并确定了一个 IPC 操作是否可以访问该 IPC 对象。

```
struct ipc_perm {
    key_t    key ;           /* 此 IPC 对象的 key 键 */
    uid_t    uid ;           /* 此 IPC 对象用户 ID */
    gid_t    gid ;           /* 此 IPC 对象组 ID */
};
```

```
uid_t    cuid ;           /* IPC 对象创建进程的有效用户 ID */
gid_t    cgid ;          /* IPC 对象创建进程的有效组 ID */
mode_t    mode ;         /* 此 IPC 的读写权限 */
ulong_t   seq ;          /* IPC 对象的序列号 */
} ;
```

表 15-2 列出了 ipc_perm 中 mode 的含义，其含义与文件访问权限相似。当调用 IPC 对象创建函数（semget、msgget、shmget）时，会对 ipc_perm 结构变量的每一个成员赋值，其中 mode 的值来源于 IPC 对象创建函数最右边的形参 flag（msgget 中为 msgflg、semget 中为 semflg、shmget 中为 shmflg）。如需修改这几个成员变量，则需调用相应的控制函数（msgctl、semctl、shmctl）。

表 15-2 IPC 对象存取权限表

ipc_perm 中 mode 的含义			
操作者	读	写	可读/可写
用户	0400	0200	0600
组	0040	0020	0060
其他	0004	0002	0006

5. IPC 对象的创建权限

msgget、semget、shmget 函数最右边的形参 flag（msgget 中为 msgflg、semget 中为 semflg、shmget 中为 shmflg）为 IPC 对象创建权限，这三种函数中 flag 的作用基本相同。

IPC 对象创建权限（即 flag）格式为 0xxxxx，其中 0 表示八进制，低三位为用户、属组，以及其他的读、写、执行权限（执行位不使用），其含义与 ipc_perm 的 mode 相同，具体含义见表 15-2。在这里姑且把 IPC 对象创建权限格式的低三位称为“IPC 对象存取权限”。如 0600 代表只有此用户下的进程才有可读、可写权限。IPC 对象存取权限常与 IPC_CREAT、IPC_EXCL 两种标志进行或（|）运算，以完成对 IPC 对象创建的管理，在这里姑且把 IPC_CREAT、IPC_EXCL 两种标志称为 IPC 创建模式标志。下面是两种创建模式标志在<sys/ipc.h>头文件中的宏定义。

```
#define IPC_CREAT    01000    /* Create key if key does not exist. */
#define IPC_EXCL     02000    /* Fail if key exists. */
```

综上所述，flag 标志由两部分组成，一是 IPC 对象存取权限（含义同 ipc_perm 中的 mode），二是 IPC 对象创建模式标志（IPC_CREAT、IPC_EXCL），两者进行或（|）运算合成 IPC 对象创建权限。

6. 创建或打开 IPC 对象流程图

semget、msgget、shmget 函数的作用是创建一个新的 IPC 对象或者访问一个已存在的 IPC 对象。其创建或访问的规则如下：

- ① 指定 key 为 IPC_PRIVATE 操作系统保证创建一个唯一的 IPC 对象。
- ② 设置 flag 参数的 IPC_CREAT 位但不设置它的 IPC_EXCL 位时，如果所指定的 key 键的 IPC 对象不存在，就创建一个新的对象；否则返回该对象。

③ 同时设置 flag 的 IPC_CREAT 和 IPC_EXCL 位时，如果所指定的 key 键的 IPC 对象不存在，就创建一个新的对象；否则返回一个 EEXIST 错误，因为该对象已存在。

综上所述，flag 创建模式标志的作用如表 15-3 所示。

表 15-3 三种xxxget 函数的 flag 创建模式标志作用表

flag 创建模式标志	不 存 在	已 存 在
无特殊标志	出错，errno=ENOENT	成功，引用已存在对象
IPC_CREAT	成功，创建新对象	成功，引用已存在对象
IPC_CREAT IPC_EXCL	成功，创建新对象	出错，errno=EEXIST

图 15-2 画出了 semget、msgget、shmget 创建或打开一个 IPC 对象的逻辑流程图，它说明了内核创建和访问 IPC 对象的流程。

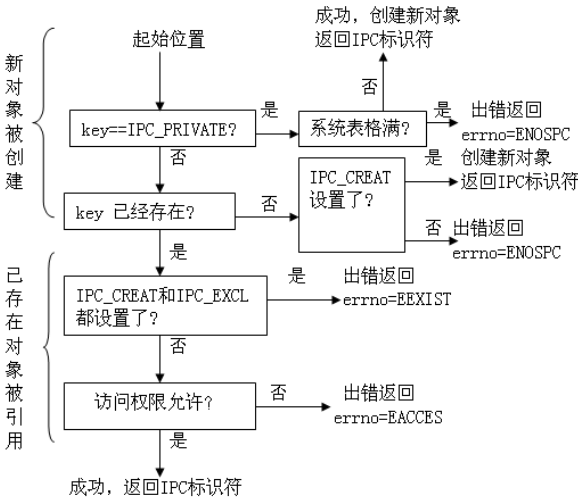


图 15-2 semget、msgget、shmget 创建或打开一个 IPC 对象的逻辑流程图

使用 semget、msgget、shmget 创建一个 IPC 对象时，需要指定 flag 标志，在 key 不等于 IPC_PRIVATE 的情况下，flag 标志决定了创建方式和创建后 IPC 对象的存取权限。在 key 等于 IPC_PRIVATE 的情况下，flag 标志决定了创建后 IPC 对象的存取权限。如果只是引用一个已经存在的 IPC 对象，只需把 flag 标志设为 0 即可。

15.2 消息队列

一个或多个进程向消息队列写入消息，另外一个或多个进程从消息队列中读取消息，这种进程间通信机制通常使用在请求/服务模型中，请求进程向服务进程发送请求的消息，服务进程读取消息并执行相应的操作。在许多微内核结构的操作系统中，内核和各组件之间的基本通信方式就是消息队列。例如，在 Minix 操作系统中，内核、I/O 任务、服务器进程和用户进程之间就是通过消息队列实现通信的。

Linux 中的消息可以被描述成在内核地址空间的一个内部链表，每一个消息队列都有唯一的

一个消息队列标识号。Linux 为系统中所有的消息队列维护一个 `msgque` 链表，该链表中的每个指针指向一个 `msqid_ds` 结构，该结构完整地描述了一个个消息队列。

15.2.1 消息队列简要说明

消息队列常使用在一个进程向消息队列发消息，而另一个进程向消息队列接收消息的场合。接收消息分两种情况，即接收最开始的一条消息和特定类型的消息。

1. 消息队列内核数据结构

当在系统中创建每一个消息队列时，内核创建、存储及维护着 `msqid_ds` 结构的一个实例。`msqid_ds` 结构的具体说明如下：

```
/* 在系统中的每一个消息队列对应一个msqid_ds 结构 */
struct msqid_ds {
    struct ipc_perm msg_perm;
    struct msg *msg_first; /* 队列中的第一条消息，即链表头 */
    struct msg *msg_last; /* 队列中的最后一条消息，即链表尾 */
    time_t msg_stime; /* 发送给队列的最后一条消息的时间 */
    time_t msg_rtime; /* 从消息队列接收到的最后一条消息的时间 */
    time_t msg_ctime; /* 最后修改队列的时间 */
    ushort msg_cbytes; /* 队列上所有消息总的字节数 */
    ushort msg_qnum; /* 在当前队列上消息的个数 */
    ushort msg_qbytes; /* 队列最大的字节数 */
    ushort msg_lspid; /* 发送最后一条消息的进程的 pid */
    ushort msg_lrpid; /* 接收最后一条消息的进程的 pid */
};
```

图 15-3 形象地画出了内核对消息队列的维护方法。内核对消息队列的管理是通过管理一个链表实现的。

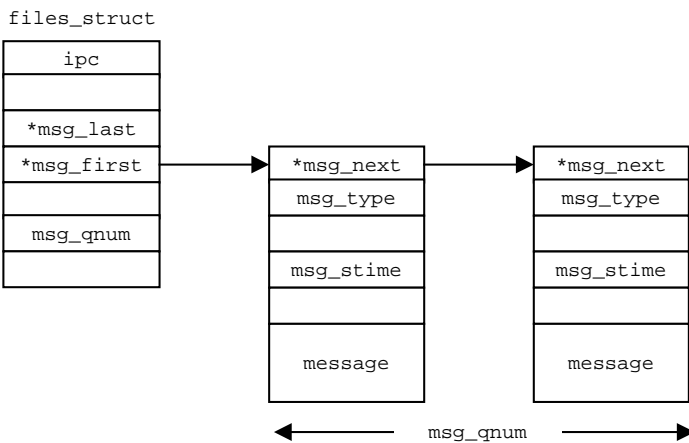


图 15-3 消息队列内核维护图

2. 消息队列通信原理图

消息队列常使用在两个进程之间收发信息的场合。图 15-4 为消息队列通信原理图，一个进

程向消息队列发送消息，而另一个进程从消息队列收取消息。

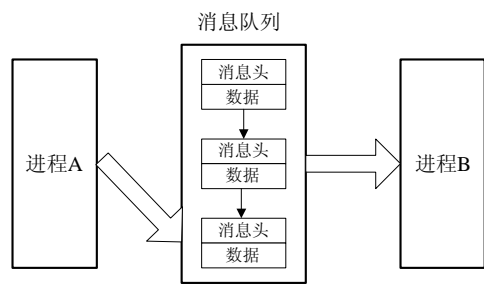


图 15-4 消息队列通信原理图

15.2.2 消息队列函数

消息队列函数由 msgget、msgctl、msgsnd、msgrcv 四个函数组成。下面介绍这四个函数的函数原型及其具体说明。

1. msgget 函数原型

msgget（得到消息队列标识符或创建一个消息队列对象）		
所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/msg.h>	
函数说明	得到消息队列标识符或创建一个消息队列对象，并返回消息队列标识符	
函数原型	int msgget(key_t key, int msgflg)	
函数传入值	key	0（IPC_PRIVATE）：会建立新的消息队列 大于 0 的 32 位整数：视参数 msgflg 来确定操作。通常要求此值来源于 ftok 返回的 IPC 键值
	msgflg	0：取消息队列标识符，若不存在，则函数会报错
		IPC_CREAT：当 msgflg&IPC_CREAT 为真时，如果内核中不存在键值与 key 相等的消息队列，则新建一个消息队列；如果存在这样的消息队列，则返回此消息队列的标识符
		IPC_CREAT IPC_EXCL：如果内核中不存在键值与 key 相等的消息队列，则新建一个消息队列；如果存在这样的消息队列，则报错
函数返回值	成功：返回消息队列的标识符	
	出错：-1，错误原因存于 error 中	
附加说明	上述 msgflg 参数为模式标志参数，使用时需要与 IPC 对象存取权限（如 0600）进行或（ ）运算来确定消息队列的存取权限	
错误代码	EACCES：指定的消息队列已存在，但调用进程没有权限访问它 EEXIST：key 指定的消息队列已存在，而 msgflg 中同时指定 IPC_CREAT 和 IPC_EXCL 标志 ENOENT：key 指定的消息队列不存在同时 msgflg 中没有指定 IPC_CREAT 标志 ENOMEM：需要建立消息队列，但内存不足 ENOSPC：需要建立消息队列，但已达到系统的限制	

如果用 msgget 创建了一个新的消息队列对象，则 msqid_ds 结构成员变量的值设置如下：

- msg_qnum、msg_lspid、msg_lrpid、msg_stime、msg_rtime 设置为 0。

- msg_ctime 设置为当前时间。
- msg_qbytes 设成系统的限制值。
- msgflg 的读写权限写入 msg_perm.mode 中。
- msg_perm 结构的 uid 和 cuid 成员被设置成当前进程的有效用户 ID，gid 和 cuid 成员被设置成当前进程的有效组 ID。

2. msgctl 函数原型

msgctl（获取和设置消息队列的属性）		
所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/msg.h>	
函数说明	获取和设置消息队列的属性	
函数原型	int msgctl(int msqid, int cmd, struct msqid_ds *buf)	
函数传入值	msqid	消息队列标识符
	cmd	IPC_STAT: 获得 msqid 的消息队列头数据到 buf 中
		IPC_SET: 设置消息队列的属性，要设置的属性需先存储在 buf 中，可设置的属性包括：msg_perm.uid、msg_perm.gid、msg_perm.mode 以及 msg_qbytes
		buf: 消息队列管理结构体，请参见消息队列内核结构说明部分
函数返回值	成功: 0	
	出错: -1, 错误原因存于 error 中	
错误代码	EACCESS: 参数 cmd 为 IPC_STAT，但无权限读取该消息队列 EFAULT: 参数 buf 指向无效的内存地址 EIDRM: 标识符为 msqid 的消息队列已被删除 EINVAL: 无效的参数 cmd 或 msqid EPERM: 参数 cmd 为 IPC_SET 或 IPC_RMID，却无足够的权限执行	

3. msgsnd 函数原型

msgsnd（将消息写入到消息队列）		
所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/msg.h>	
函数说明	将 msgp 消息写入标识符为 msqid 的消息队列	
函数原型	int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg)	
函数传入值	msqid	消息队列标识符
	msgp	发送给队列的消息。msgp 可以是任何类型的结构体，但第一个字段必须为 long 类型，即表明此发送消息的类型，msgrcv 根据此接收消息。msgp 定义的参照格式如下： <pre>struct s_msg{ /*msgp 定义的参照格式*/ long type; /* 必须大于 0，消息类型 */ char mtext[256]; /*消息正文，可以是其他任何类型*/ } msgp;</pre>

续表

msgsnd（将消息写入消息队列）		
函数传入值	msgsz	要发送消息的大小，不含消息类型占用的 4 个字节，即 mtext 的长度
	msgflg	0：当消息队列满时，msgsnd 将会阻塞，直到消息能写入消息队列
		IPC_NOWAIT：当消息队列已满的时候，msgsnd 函数不等待立即返回
		IPC_NOERROR：若发送的消息大于 size 字节，则把该消息截断，截断部分将被丢弃，且不通知发送进程
函数返回值	成功：0	
	出错：-1，错误原因存于 error 中	
错误代码	EAGAIN：参数 msgflg 设为 IPC_NOWAIT，而消息队列已满 EIDRM：标识符为 msqid 的消息队列已被删除 EACCESS：无权限写入消息队列 EFAULT：参数 msgp 指向无效的内存地址 EINTR：队列已满而处于等待情况下被信号中断 EINVAL：无效的参数 msqid、msgsz 或参数消息类型 type 小于 0	

msgsnd() 为阻塞函数，当消息队列容量满或消息个数满会阻塞。消息队列已被删除，则返回 EIDRM 错误；被信号中断返回 E_INTR 错误。

如果设置 IPC_NOWAIT 消息队列满或个数满时会返回-1，并且置 EAGAIN 错误。

msgsnd()解除阻塞的条件有以下三个条件：

- ① 不满足消息队列满或个数满两个条件，即消息队列中有容纳该消息的空间。
- ② sqid 代表的消息队列被删除。
- ③ 调用 msgsnd 函数的进程被信号中断。

4. msgrcv 函数原型

msgrcv（从消息队列读取消息）		
所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/msg.h>	
函数说明	从标识符为 msqid 的消息队列读取消息并存于 msgp 中，读取后把此消息从消息队列中删除	
函数原型	ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);	
函数传入值	msqid	消息队列标识符
	msgp	存放消息的结构体，结构体类型要与 msgsnd 函数发送的类型相同
	msgsz	要接收消息的大小，不含消息类型占用的 4 个字节
	msgtyp	0：接收第一个消息
		>0：接收类型等于 msgtyp 的第一个消息
		<0：接收类型等于或者小于 msgtyp 绝对值的第一个消息
	msgflg	0：阻塞式接收消息，没有该类型的消息 msgrcv 函数一直阻塞等待
		IPC_NOWAIT：如果没有返回条件的消息调用立即返回，此时错误码为 ENMSG
		IPC_EXCEPT：与 msgtype 配合使用返回队列中第一个类型不为 msgtype 的消息
		IPC_NOERROR：如果队列中满足条件的消息内容大于所请求的 size 字节，则把该消息截断，截断部分将被丢弃

续表

msgrcv (从消息队列读取消息)	
函数返回值	成功: 实际读取到的消息数据长度
	出错: -1, 错误原因存于 error 中
错误代码	E2BIG: 消息数据长度大于 msgsz, 而 msgflag 没有设置 IPC_NOERROR
	EIDRM: 标识符为 msqid 的消息队列已被删除
	EACCESS: 无权限读取该消息队列
	EFAULT: 参数 msgp 指向无效的内存地址
	ENOMSG: 参数 msgflg 设为 IPC_NOWAIT, 而消息队列中无消息可读
	EINTR: 等待读取队列内的消息情况下被信号中断

msgrcv()解除阻塞的条件有以下三个:

- ① 消息队列中有了满足条件的消息。
- ② msqid 代表的消息队列被删除。
- ③ 调用 msgrcv()的进程被信号中断。

15.2.3 消息队列使用程序范例

1. 消息队列控制范例

msgctl.c 源代码如下:

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <error.h>
#define TEXT_SIZE 512
struct msgbuf
{
    long mtype ;
    char mtext[TEXT_SIZE] ;
} ;
int main(int argc, char **argv)
{
    int msqid ;
    struct msqid_ds info ;
    struct msgbuf buf ;
    struct msgbuf buf1 ;
    int flag ;
    int sendlength, recvlength ;

    msqid = msgget( IPC_PRIVATE, 0666 ) ;
    if ( msqid < 0 )
    {
        perror("get ipc_id error") ;
        return -1 ;
    }
```

```

    }

    buf.mtype = 1 ;
    strcpy(buf.mtext, "happy new year!") ;
    sendlength = sizeof(struct msgbuf) - sizeof(long) ;
    flag = msgsnd( msqid, &buf, sendlength , 0 ) ;
    if ( flag < 0 )
    {
        perror("send message error") ;
        return -1 ;
    }
    buf.mtype = 3 ;
    strcpy(buf.mtext, "good bye!") ;
    sendlength = sizeof(struct msgbuf) - sizeof(long) ;
    flag = msgsnd( msqid, &buf, sendlength , 0 ) ;
    if ( flag < 0 )
    {
        perror("send message error") ;
        return -1 ;
    }

    flag = msgctl( msqid, IPC_STAT, &info ) ;
    if ( flag < 0 )
    {
        perror("get message status error") ;
        return -1 ;
    }
    printf("uid:%d, gid = %d, cuid = %d, cgid= %d\n" ,
        info.msg_perm.uid, info.msg_perm.gid, info.msg_perm.cuid, info.msg_
perm.cgid ) ;
    printf("read-write:%03o, cbytes = %lu, qnum = %lu, qbytes= %lu\n" ,
        info.msg_perm.mode&0777, info.msg_cbytes, info.msg_qnum, info.msg_qbytes);
    system("ipcs -q") ;
    recvlength = sizeof(struct msgbuf) - sizeof(long) ;
    memset(&buf1, 0x00, sizeof(struct msgbuf)) ;

    flag = msgrcv( msqid, &buf1, recvlength ,3,0 ) ;
    if ( flag < 0 )
    {
        perror("recv message error") ;
        return -1 ;
    }
    printf("type=%d, message=%s\n", buf1.mtype, buf1.mtext) ;

    flag = msgctl( msqid, IPC_RMID,NULL) ;
    if ( flag < 0 )
    {
        perror("rm message queue error") ;
        return -1 ;
    }
    system("ipcs -q") ;

    return 0 ;
}

```

编译 gcc msgctl.c -o msgctl。

执行 ./msg, 执行结果如下:

```
uid:1008, gid = 1003, cuid = 1008, cgid= 1003
read-write:666, cbytes = 1024, qnum = 2, qbytes= 163840

----- Message Queues -----
key          msqid      owner      perms      used-bytes   messages
0x00000000  65536          zjkf       666        1024         2

type=3, message=good bye!

----- Message Queues -----
key          msqid      owner      perms      used-bytes   messages
```

2. 两进程间通过消息队列收发消息

(1) 发送消息队列程序

msgsnd.c 源代码如下:

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <time.h>
#define TEXT_SIZE 512
struct msgbuf
{
    long mtype ;
    int status ;
    char time[20] ;
    char mtext[TEXT_SIZE] ;
} ;
char *getxtsj()
{
    time_t tv ;
    struct tm *tmp ;
    static char buf[20] ;
    tv = time( 0 ) ;
    tmp = localtime(&tv) ;
    sprintf(buf,"%02d:%02d:%02d",tmp->tm_hour , tmp->tm_min,tmp->tm_sec);
    return buf ;
}
int main(int argc, char **argv)
{
    int msqid ;
    struct msqid_ds info ;
    struct msgbuf buf ;
    struct msgbuf buf1 ;
    int flag ;
    int sendlength, recvlenght ;
    int key ;

    key = ftok("msg.tmp", 0x01 ) ;
```

```

if ( key < 0 )
{
    perror("ftok key error") ;
    return -1 ;
}

msqid = msgget( key, 0600|IPC_CREAT ) ;
if ( msqid < 0 )
{
    perror("create message queue error") ;
    return -1 ;
}

buf.mtype = 1 ;
buf.status = 9 ;
strcpy(buf.time, getxtime()) ;
strcpy(buf.mtext, "happy new year!") ;
sendlength = sizeof(struct msgbuf) - sizeof(long) ;
flag = msgsnd( msqid, &buf, sendlength , 0 ) ;
if ( flag < 0 )
{
    perror("send message error") ;
    return -1 ;
}
buf.mtype = 3 ;
buf.status = 9 ;
strcpy(buf.time, getxtime()) ;
strcpy(buf.mtext, "good bye!") ;
sendlength = sizeof(struct msgbuf) - sizeof(long) ;
flag = msgsnd( msqid, &buf, sendlength , 0 ) ;
if ( flag < 0 )
{
    perror("send message error") ;
    return -1 ;
}
system("ipcs -q") ;
return 0 ;
}

```

(2) 接收消息队列程序

msgrcv.c 源代码如下:

```

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define TEXT_SIZE 512
struct msgbuf
{
    long mtype ;
    int status ;
    char time[20] ;
    char mtext[TEXT_SIZE] ;
} ;

```

```
int main(int argc, char **argv)
{
    int msqid ;
    struct msqid_ds info ;
    struct msgbuf buf1 ;
    int flag ;
    int recvlength ;
    int key ;
    int mtype ;

    key = ftok("msg.tmp", 0x01 ) ;
    if ( key < 0 )
    {
        perror("ftok key error") ;
        return -1 ;
    }

    msqid = msgget( key, 0 ) ;
    if ( msqid < 0 )
    {
        perror("get ipc_id error") ;
        return -1 ;
    }

    recvlength = sizeof(struct msgbuf) - sizeof(long) ;
    memset(&buf1, 0x00, sizeof(struct msgbuf)) ;
    mtype = 1 ;
    flag = msgrcv( msqid, &buf1, recvlength ,mtype,0 ) ;
    if ( flag < 0 )
    {
        perror("recv message error\n") ;
        return -1 ;
    }
    printf("type=%d,time=%s, message=%s\n", buf1.mtype, buf1.time, buf1.mtext) ;
    system("ipcs -q") ;
    return 0 ;
}
```

(3) 编译与执行程序

- ① 在当前目录下利用>msg.tmp 建立空文件 msg.tmp。
- ② 编译发送消息队列程序 gcc msgsnd.c -o msgsnd。
- ③ 执行./msgsnd，执行结果如下：

```
----- Message Queues -----
```

key	msqid	owner	perms	used-bytes	messages
0x0101436d	294912	zjkf	600	1072	2

- ④ 编译接收消息程序 gcc msgrcv.c -o msgrcv。

- ⑤ 执行./msgrcv，执行结果如下：

type=1,time=03:23:16, message=happy new year!

```
----- Message Queues -----
```

key	msqid	owner	perms	used-bytes	messages
0x0101436d	294912	zjkf	600	536	1

⑥ 利用 `ipcrm -q 294912` 删除该消息队列。因为消息队列是随内核持续存在的，在程序中若不利用 `msgctl` 函数或在命令行用 `ipcrm` 命令显式地删除，该消息队列就一直存在于系统中。另外，信号量和共享内存也是随内核持续存在的。

15.3 信号量

15.3.1 信号量简要说明

信号量（也称信号灯）与其他进程间通信的方式不大相同，它主要提供对进程间共享资源访问控制机制，相当于内存中的标志，进程可以根据它判定是否能够访问某些共享资源，从而实现多个进程对某些共享资源的互斥访问；同时，进程也可以修改该标志。信号量除了用于访问控制外，还可用于进程同步。由于一个信号量标识符指向的是一组信号量，所以，在这里把信号量称为信号量集，一个信号量集使用同一个信号量标识符（或称信号量集标识符），管理的是一组信号量。这样实现避免了系统中有过多的信号量对象，而且易于编程。用户使用信号量时以信号量集中的每一个信号量为操作单位，所以，操作信号量时要指明信号量标识符和该信号量在信号量集中的编号。后文为了避免理解歧义，将信号量标识符统一称为信号量集标识符。

1. 信号量集内核图

图 15-5 展示了信号量集的实现方法。从图中可以看出，`sem_base` 指向的是一组信号量，所以一个信号量集管理的是一组信号量，有关该信号量集中信号量的个数，用户可根据实际需要进行自行指定。

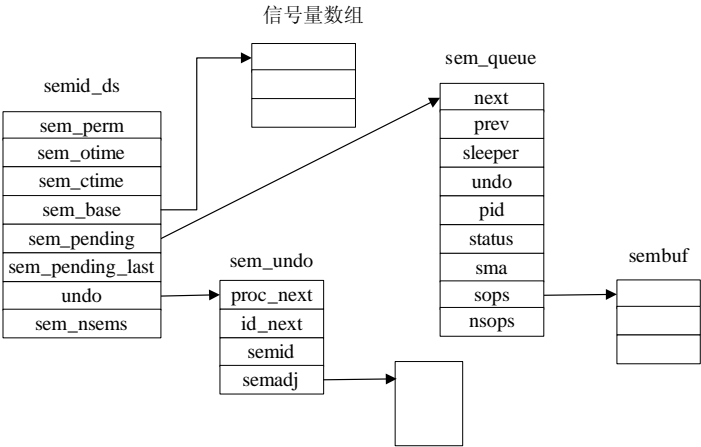


图 15-5 信号量集内核图

2. 信号量集内核结构定义

在 Linux 系统中，内核对信号量整体的管理是通过 `semary` 数组来实现的，`semary` 变量在系统中的定义如下：

```
struct semid_ds *semary[SEMMNI];
```

semary 数组中的每个元素都是一个指向 semid_ds 数据结构的指针，而一个 semid_ds 数据结构则描述了一个信号量集。SEMMNI 的值是 128，它限制了系统中同时存在的信号量集的数量，但一个信号量集可以管理一组信号量，所以，这个数值远远够用。

semid_ds 结构定义如下：

```
struct semid_ds {
    struct ipc_perm sem_perm;           /* 包含信号量集资源的属主和访问权限 */
    __kernel_time_t sem_otime;         /* 最后一次操作的时间 */
    __kernel_time_t sem_ctime;         /* 最后一次修改的时间 */
    struct sem *sem_base;              /* 指向信号量集合的指针 */
    struct sem_queue *sem_pending;      /* 挂起信号量操作队列 */
    struct sem_queue **sem_pending_last; /* 最后挂起信号量操作队列 */
    struct sem_undo *undo;              /* undo 标志信号量列表指针 */
    unsigned short sem_nsems;          /* 此信号量集中信号量的个数 */
};

struct sem {
    unsigned short semval;              /* 当前信号量值 */
    pid_t sempid;                      /* 最后修改信号量的进程 */
    unsigned short semcnt;              /* 等待进行 P 操作的进程数 */
    unsigned short semzcnt;             /* 等待 semval 为 0 的进程数 */
}
```

15.3.2 信号量函数

信号量函数由 semget、semop、semctl 三个函数组成。下面介绍这三个函数的函数原型及具体说明。

1. semget 函数原型

semget(得到一个信号量集标识符或创建一个信号量集对象)		
所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h>	
函数说明	得到一个信号量集标识符或创建一个信号量集对象，并返回信号量集标识符	
函数原型	int semget(key_t key, int nsems, int semflg)	
函数传入值	key	0 (IPC_PRIVATE): 会建立新信号量集对象 大于 0 的 32 位整数: 视参数 semflg 来确定操作，通常要求此值来源于 ftok 返回的 IPC 键值
	nsems	创建信号量集中信号量的个数，该参数只在创建信号量集时有效
	msgflg	0: 取信号量集标识符，若不存在，则函数会报错
		IPC_CREAT: 当 semflg&IPC_CREAT 为真时，如果内核中不存在键值与 key 相等的信号量集，则新建一个信号量集；如果存在这样的信号量集，则返回此信号量集的标识符
函数返回值	IPC_CREAT IPC_EXCL: 如果内核中不存在键值与 key 相等的信号量集，则新建一个消息队列；如果存在这样的信号量集，则报错	
	成功: 返回信号量集的标识符 出错: -1, 错误原因存于 error 中	
附加说明	上述 semflg 参数为模式标志参数，使用时需要与 IPC 对象存取权限（如 0600）进行或（ ）运算来确定信号量集的存取权限	

续表

semget(得到一个信号量集标识符或创建一个信号量集对象)	
错误代码	<div>EACCESS: 没有权限</div> <div>EEXIST: 信号量集已经存在, 无法创建</div> <div>EIDRM: 信号量集已经删除</div> <div>ENOENT: 信号量集不存在, 同时 semflg 没有设置 IPC_CREAT 标志</div> <div>ENOMEM: 没有足够的内存创建新的信号量集</div> <div>ENOSPC: 超出限制</div>

如果用 semget 创建了一个新的信号量集对象, 则 semid_ds 结构成员变量的值设置如下:

- sem_otime 设置为 0。
- sem_ctime 设置为当前时间。
- msg_qbytes 设成系统的限制值。
- sem_nsems 设置为 nsems 参数的数值。
- semflg 的读写权限写入 sem_perm.mode 中。
- sem_perm 结构的 uid 和 cuid 成员被设置成当前进程的有效用户 ID, gid 和 cuid 成员被设置成当前进程的有效组 ID。

2. semop 函数原型

semop (完成对信号量的 P 操作或 V 操作)	
所需头文件	<div>#include <sys/types.h></div> <div>#include <sys/ipc.h></div> <div>#include <sys/sem.h></div>
函数说明	对信号量集标识符为 semid 中的一个或多个信号量进行 P 操作或 V 操作
函数原型	int semop(int semid, struct sembuf *sops, unsigned nsops)
函数传入值	semid: 信号量集标识符
	<div>sops: 指向进行操作的信号量集结构体数组的首地址, 此结构的具体说明如下:</div> <div>struct sembuf {</div> <div> short semnum; /*信号量集中的信号量编号, 0 代表第 1 个信号量*/</div> <div> short val; /*若 val>0, 进行 V 操作信号量值加 val, 表示进程释放控制的资源 */</div> <div> /*若 val<0, 进行 P 操作信号量值减 val, 若 (semval-val)<0 (semval 为该信号量值), 则调用进程阻塞, 直到资源可用; 若设置 IPC_NOWAIT 不会睡眠, 进程直接返回 EAGAIN 错误*/</div> <div> /*若 val==0 时, 阻塞等待信号量为 0, 调用进程进入睡眠状态, 直到信号值为 0; 若设置 IPC_NOWAIT, 进程不会睡眠, 直接返回 EAGAIN 错误*/</div> <div> short flag; /*0 设置信号量的默认操作*/</div> <div> /*IPC_NOWAIT 设置信号量操作不等待*/</div> <div> /*SEM_UNDO 选项会让内核记录一个与调用进程相关的 UNDO 记录, 如果该进程崩溃, 则根据这个进程的 UNDO 记录自动恢复相应信号量的计数值*/</div> <div>};</div>
	nsops: 进行操作信号量的个数, 即 sops 结构变量的个数, 需大于或等于 1。最常见的设置是此值等于 1, 只完成对一个信号量的操作

续表

semop（完成对信号量的 P 操作或 V 操作）	
函数返回值	成功：返回信号量集的标识符
	出错：-1，错误原因存于 error 中
错误代码	E2BIG：一次对信号量个数的操作超过了系统限制 EACCESS：权限不够 EAGAIN：使用了 IPC_NOWAIT，但操作不能继续进行 EFAULT：sops 指向的地址无效 EIDRM：信号量集已经删除 EINTR：当睡眠时接收到其他信号 EINVAL：信号量集不存在，或者 semid 无效 ENOMEM：使用了 SEM_UNDO，但无足够的内存创建所需的数据结构 ERANGE：信号量值超出范围

上文 semop 函数中 sops 形参为指向 sembuf 数组的指针，该结构体数组定义信号量要进行的操作序列。下面是其操作定义举例。

```
struct sembuf sem_get={0,-1,IPC_NOWAIT}; /*将信号量对象中序号为 0 的信号量减 1*/
struct sembuf sem_get={0,1,IPC_NOWAIT}; /*将信号量对象中序号为 0 的信号量加 1*/
struct sembuf sem_get={0,0,0};          /*进程被阻塞，直到对应的信号量值为 0*/
```

sembuf 结构体 flag 成员一般为 0，若 flag 包含 IPC_NOWAIT，则该操作为非阻塞操作。若 flag 包含 SEM_UNDO，则当进程退出的时候会还原该进程的信号量操作，这个标志在某些情况下是很有用的，比如，某进程做了 P 操作得到资源，但还没来得及做 V 操作时就异常退出了，此时，其他进程就只能都阻塞在 P 操作上，于是造成了死锁。若采取 SEM_UNDO 标志，就可以避免因为进程异常退出而造成的死锁。

3. semctl 函数原型

semctl（得到一个信号量集标识符或创建一个信号量集对象）		
所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h>	
函数说明	得到一个信号量集标识符或创建一个信号量集对象，并返回信号量集标识符	
函数原型	int semctl(int semid, int semnum, int cmd, union semun arg)	
函数传入值	semid	信号量集标识符
	semnum	信号量集数组上的下标，表示某一个信号量
	cmd	如表 15-4 所示
	arg	union semun { short val; /*SETVAL 用的值*/ struct semid_ds* buf; /*IPC_STAT、IPC_SET 用的 semid_ds 结构*/ unsigned short* array; /*SETALL、GETALL 用的数组值*/ struct seminfo *buf; /*为控制 IPC_INFO 提供的缓存*/ } arg;
函数返回值	成功：大于或等于 0，具体说明请参照表 15-4	
	出错：-1，错误原因存于 error 中	

续表

semctl (得到一个信号量集标识符或创建一个信号量集对象)	
附加说明	semid_ds 结构见上文信号量集内核结构定义
错误代码	EACCESS: 权限不够 EFAULT: arg 指向的地址无效 EIDRM: 信号量集已经删除 EINVAL: 信号量集不存在, 或者 semid 无效 EPERM: 进程有效用户没有 cmd 的权限 ERANGE: 信号量值超出范围

表 15-4 semctl 函数 cmd 形参说明表

命 令	解 释
IPC_STAT	从信号量集上检索 semid_ds 结构, 并存到 semun 联合体参数的成员 buf 的地址中
IPC_SET	设置一个信号量集合的 semid_ds 结构中 ipc_perm 域的值, 并从 semun 的 buf 中取出值
IPC_RMID	从内核中删除信号量集合
GETALL	从信号量集合中获得所有信号量的值, 并把其整数值存到 semun 联合体成员的一个指针数组中
GETNCNT	返回当前等待资源的进程个数
GETPID	返回最后一个执行系统调用 semop() 进程的 PID
GETVAL	返回信号量集合内单个信号量的值
GETZCNT	返回当前等待 100% 资源利用的进程个数
SETALL	与 GETALL 正好相反
SETVAL	用联合体中 val 成员的值设置信号量集合中单个信号量的值

15.3.3 信号量应用程序示例

sem.c 源代码如下:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
union semun {
    int val; /* value for SETVAL */
    struct semid_ds *buf; /* buffer for IPC_STAT, IPC_SET */
    unsigned short *array; /* array for GETALL, SETALL */
    struct seminfo *__buf; /* buffer for IPC_INFO */
};
/**对信号量数组 semnum 编号的信号量做 P 操作***/
int P(int semid, int semnum)
{
    struct sembuf sops={semnum,-1, SEM_UNDO};
    return (semop(semid,&sops,1));
}
/**对信号量数组 semnum 编号的信号量做 V 操作***/
int V(int semid, int semnum)
{
    struct sembuf sops={semnum,+1, SEM_UNDO};
    return (semop(semid,&sops,1));
}
```

```

}

int main(int argc, char **argv)
{
    int key ;
    int semid,ret;
    union semun arg;
    struct sembuf semop;
    int flag ;

    key = ftok("/tmp", 0x66 ) ;
    if ( key < 0 )
    {
        perror("ftok key error") ;
        return -1 ;
    }
    /**本程序创建了三个信号量，实际使用时只用了一个 0 号信号量***/
    semid = semget(key,3,IPC_CREAT|0600);
    if (semid == -1)
    {
        perror("create semget error");
        return ;
    }
    if ( argc == 1 )
    {
        arg.val = 1;
        /**对 0 号信号量设置初始值***/
        ret =semctl(semid,0,SETVAL,arg);
        if (ret < 0 )
        {
            perror("ctl sem error");
            semctl(semid,0,IPC_RMID,arg);
            return -1 ;
        }
    }
    /**取 0 号信号量的值***/
    ret =semctl(semid,0,GETVAL,arg);
    printf("after semctl setval  sem[0].val =[%d]\n",ret);
    system("date") ;
    printf("P operate begin\n") ;
    flag = P(semid,0) ;
    if ( flag )
    {
        perror("P operate error") ;
        return -1 ;
    }
    printf("P operate end\n") ;
    ret =semctl(semid,0,GETVAL,arg);
    printf("after P sem[0].val=[%d]\n",ret);
    system("date") ;
    if ( argc == 1 )
    {
        sleep(120) ;
    }
    printf("V operate begin\n") ;

```

```
if (V(semid, 0) < 0)
{
    perror("V operate error") ;
    return -1 ;
}
printf("V operate end\n") ;
ret =semctl(semid,0,GETVAL,arg);
printf("after V sem[0].val=%d\n",ret);
system("date") ;
if ( argc >1 )
{
    semctl(semid,0,IPC_RMID,arg);
}

return 0 ;
}
```

① 编译 `gcc sem.c -o sem`。

② 在窗口中执行 `./sem`，执行结果如下：

```
after semctl setval sem[0].val =[1]
2011 年 01 月 11 日 星期二 10:08:11 CST
P operate begin
P operate end
after P sem[0].val=[0]
2011 年 01 月 11 日 星期二 10:08:11 CST
V operate begin
V operate end
after V sem[0].val=0
2011 年 01 月 11 日 星期二 10:10:11 CST
```

③ 在另一个窗口中执行 `./sem test1`，执行结果如下：

```
after semctl setval sem[0].val =[0]
2011 年 01 月 11 日 星期二 10:08:36 CST
P operate begin
P operate end
after P sem[0].val=[0]
2011 年 01 月 11 日 星期二 10:10:11 CST
V operate begin
V operate end
after V sem[0].val=1
2011 年 01 月 11 日 星期二 10:10:11 CST
```

15.4 共享内存

15.4.1 共享内存简要说明

共享内存区域是被多个进程共享的一部分物理内存。如果多个进程都把该内存区域映射到自己的虚拟地址空间，则这些进程就都可以直接访问该共享内存区域，从而可以通过该区域进行通信。共享内存是进程间共享数据的一种最快的方法，一个进程向共享内存区域写入了数据，共享

这个内存区域的所有进程就可以立刻看到其中的内容。这块共享虚拟内存的页面出现在每一个共享该页面的进程的页表中。图 15-6 为共享内存实现原理图。

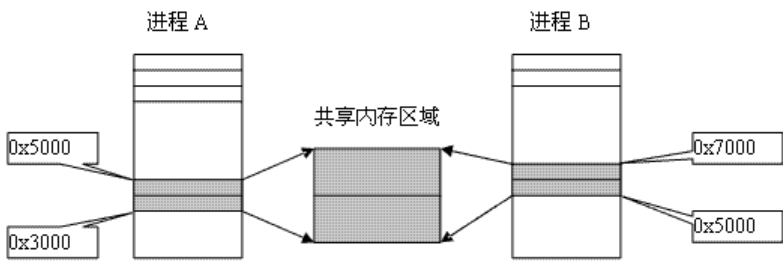


图 15-6 共享内存实现原理图

1. 共享内存内核结构定义

每一个新创建的共享内存对象都可用一个 `shmid_kernel` 数据结构来表达。系统中所有的 `shmid_kernel` 数据结构都保存在 `shm_segs` 数组中，`shm_segs` 数组的每一个元素都是一个指向 `shmid_kernel` 数据结构的指针，`shm_segs` 数组管理着系统中所有的共享内存。`shmid_kernel` 结构和 `shm_segs` 变量的定义如下：

```
struct shmid_kernel *shm_segs[SHMMNI];
/* SHMMNI 为 128，表示系统中最多可以有 128 个共享内存对象 */
/* 数据结构 shmid_kernel 的定义如下： */
struct shmid_kernel
{
    struct shmid_ds u;          /* the following are private */
    unsigned long shm_npages;   /* size of segment (pages) */
    unsigned long *shm_pages;   /* array of ptrs to frames->SHMMAX */
    struct vm_area_struct *attaches; /* descriptors for attaches */
};
```

其中：

- `shm_pages`：是指向该共享内存对象所占据的内存页面数组，数组中的每个元素是每个内存页面的起始地址。
- `shm_npages`：是该共享内存区域的大小，以页为单位。
- `shmid_ds`：是一个数据结构，它描述了这个共享内存区的访问权限信息，字节大小，最后一次粘附时间、分离时间、改变时间，创建该共享区域的进程 ID，最后一次对它操作的进程 ID，当前有多少个进程在使用它的信息等。其定义如下：

```
struct shmid_ds {
    struct ipc_perm shm_perm; /* operation perms */
    int shm_segsz;           /* size of segment (bytes) */
    __kernel_time_t shm_atime; /* last attach time */
    __kernel_time_t shm_dtime; /* last detach time */
    __kernel_time_t shm_ctime; /* last change time */
    __kernel_ipc_pid_t shm_cpid; /* pid of creator */
    __kernel_ipc_pid_t shm_lpid; /* pid of last operator */
    unsigned short shm_nattch; /* number of current attaches */
};
```

```
    unsigned short shm_unused;    /* compatibility */
    void *shm_unused2;           /* ditto - used by DIPC */
    void *shm_unused3;           /* unused */
};
```

15.4.2 共享内存函数

共享内存函数由 shmget、shmat、shmdt、shmctl 四个函数组成。下面介绍这四个函数的函数原型及其具体说明。

1. shmget 函数原型

shmget(得到一个共享内存标识符或创建一个共享内存对象)		
所需头文件	#include <sys/ipc.h> #include <sys/shm.h>	
函数说明	得到一个共享内存标识符或创建一个共享内存对象，并返回共享内存标识符	
函数原型	int shmget(key_t key, size_t size, int shmflg)	
函数传入值	key	0 (IPC_PRIVATE): 会建立新共享内存对象
		大于 0 的 32 位整数: 视参数 shmflg 来确定操作。通常要求此值来源于 ftok 返回的 IPC 键值
	size	大于 0 的整数: 新建的共享内存大小，以字节为单位
		0: 只获取共享内存时指定为 0
	shmflg	0: 取共享内存标识符，若不存在，则函数会报错
		IPC_CREAT: 当 shmflg&IPC_CREAT 为真时，如果内核中不存在键值与 key 相等的共享内存，则新建一个共享内存；如果存在这样的共享内存，返回此共享内存的标识符
		IPC_CREAT IPC_EXCL: 如果内核中不存在键值与 key 相等的共享内存，则新建一个消息队列；如果存在这样的共享内存，则报错
函数返回值	成功: 返回共享内存的标识符	
	出错: -1, 错误原因存于 error 中	
附加说明	上述 shmflg 参数为模式标志参数，使用时需要与 IPC 对象存取权限（如 0600）进行 运算来确定信号量集的存取权限	
错误代码	EINVAL: 参数 size 小于 SHMMIN 或大于 SHMMAX EEXIST: 预建立 key 所指的共享内存，但已经存在 EIDRM: 参数 key 所指的共享内存已经删除 ENOSPC: 超过了系统允许建立的共享内存的最大值（SHMALL） ENOENT: 参数 key 所指的共享内存不存在，而参数 shmflg 未设 IPC_CREAT 位 EACCES: 没有权限 ENOMEM: 核心内存不足	

- 在 Linux 环境中，对开始申请的共享内存空间进行了初始化，初始值为 0x00。
- 如果用 shmget 创建了一个新的消息队列对象，则 shmid_ds 结构成员变量的值设置如下：
- shm_lpid、shm_nattach、shm_atime、shm_dtime 设置为 0。
 - msg_ctime 设置为当前时间。
 - shm_segsz 设成创建共享内存的大小。

- shmflg 的读写权限放在 shm_perm.mode 中。
- shm_perm 结构的 uid 和 cuid 成员被设置成当前进程的有效用户 ID，gid 和 cuid 成员被设置成当前进程的有效组 ID。

2. shmat 函数原型

shmat（把共享内存区对象映射到调用进程的地址空间）		
所需头文件	#include <sys/types.h> #include <sys/shm.h>	
函数说明	连接共享内存标识符为 shmid 的共享内存，连接成功后，把共享内存区对象映射到调用进程的地址空间，随后可像访问本地空间一样访问它	
函数原型	void *shmat(int shmid, const void *shmaddr, int shmflg)	
函数传入值	msqid	共享内存标识符
	shmaddr	指定共享内存出现在进程内存地址的什么位置，直接指定为 NULL 让内核自己决定一个合适的地址位置
	shmflg	SHM_RDONLY: 为只读模式，其他为读写模式
函数返回值	成功：连接好的共享内存地址	
	出错：-1，错误原因存于 error 中	
附加说明	fork 后子进程继承已连接的共享内存地址。exec 后该子进程与已连接的共享内存地址自动脱离(detach)。进程结束后，已连接的共享内存地址会自动脱离(detach)	
错误代码	EACCES: 无权限以指定方式连接共享内存 EINVAL: 无效的参数 shmid 或 shmaddr ENOMEM: 核心内存不足	

3. shmdt 函数原型

shmat（断开共享内存连接）	
所需头文件	#include <sys/types.h> #include <sys/shm.h>
函数说明	与 shmat 函数相反，用来断开与共享内存附加点的地址，禁止本进程访问此片共享内存
函数原型	int shmdt(const void *shmaddr)
函数传入值	shmaddr: 连接的共享内存的起始地址
函数返回值	成功：0
	出错：-1，错误原因存于 error 中
附加说明	本函数调用并不删除所指定的共享内存区，而只是将先前用 shmat 函数连接(attach)好的共享内存脱离(detach)目前的进程
错误代码	EINVAL: 无效的参数 shmaddr

4. shmctl 函数原型

shmctl（共享内存管理）	
所需头文件	#include <sys/types.h> #include <sys/shm.h>
函数说明	完成对共享内存的控制
函数原型	int shmctl(int shmid, int cmd, struct shmids *buf)

续表

shmctl (共享内存管理)		
函数传入值	msqid	共享内存标识符
	cmd	IPC_STAT: 得到共享内存的状态, 把共享内存的 shmid_ds 结构复制到 buf 中
		IPC_SET: 改变共享内存的状态, 把 buf 所指的 shmid_ds 结构中的 uid、gid、mode 复制到共享内存的 shmid_ds 结构内
		IPC_RMID: 删除这片共享内存
	buf	共享内存管理结构体。具体说明参见共享内存内核结构定义部分
函数返回值	成功: 0	
	出错: -1, 错误原因存于 error 中	
错误代码	EACCESS: 参数 cmd 为 IPC_STAT, 但无权限读取该共享内存 EFAULT: 参数 buf 指向无效的内存地址 EIDRM: 标识符为 msqid 的共享内存已被删除 EINVAL: 无效的参数 cmd 或 shmid EPERM: 参数 cmd 为 IPC_SET 或 IPC_RMID, 却无足够的权限执行	

15.4.3 共享内存应用范例

1. 父子进程通信范例

父子进程通信范例, shm.c 源代码如下:

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <error.h>
#define SIZE 1024
int main()
{
    int shmid ;
    char *shmaddr ;
    struct shmid_ds buf ;
    int flag = 0 ;
    int pid ;

    shmid = shmget(IPC_PRIVATE, SIZE, IPC_CREAT|0600 ) ;
    if ( shmid < 0 )
    {
        perror("get shm ipc_id error") ;
        return -1 ;
    }
    pid = fork() ;
    if ( pid == 0 )
    {
        shmaddr = (char *)shmat( shmid, NULL, 0 ) ;
        if ( (int)shmaddr == -1 )
        {
            perror("shmat addr error") ;
```



```

        return -1 ;

    }
    strcpy( shmaddr, "Hi, I am child process!\n" ) ;
    shmdt( shmaddr ) ;
    return 0;
} else if ( pid > 0 ) {
    sleep(3 ) ;
    flag = shmctl( shmid, IPC_STAT, &buf ) ;
    if ( flag == -1 )
    {
        perror("shmctl shm error") ;
        return -1 ;
    }

    printf("shm_segsz =%d bytes\n", buf.shm_segsz ) ;
    printf("parent pid=%d, shm_cpid = %d \n", getpid(), buf.shm_cpid ) ;
    printf("chlid pid=%d, shm_lpid = %d \n",pid , buf.shm_lpid ) ;
    shmaddr = (char *) shmat(shmid, NULL, 0 ) ;
    if ( (int)shmaddr == -1 )
    {
        perror("shmat addr error") ;
        return -1 ;
    }
    printf("%s", shmaddr) ;
    shmdt( shmaddr ) ;
    shmctl(shmid, IPC_RMID, NULL) ;
} else {
    perror("fork error") ;
    shmctl(shmid, IPC_RMID, NULL) ;
}

return 0 ;
}

```

编译 `gcc shm.c -o shm`。

执行 `./shm`，执行结果如下：

```

shm_segsz =1024 bytes
shm_cpid = 9503
shm_lpid = 9504
Hi, I am child process!

```

2. 多进程读写范例

多进程读写即一个进程写共享内存，一个或多个进程读共享内存。下面的例子实现的是一个进程写共享内存，另一个进程读共享内存。

(1) 下列程序实现了创建共享内存，并写入消息

`shmwrite.c` 源代码如下：

```

#include <stdio.h>
#include <sys/ipc.h>

```

```

#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
typedef struct{
    char name[8];
    int age;
} people;
int main(int argc, char** argv)
{
    int shm_id,i;
    key_t key;
    char temp[8];
    people *p_map;
    char pathname[30] ;

    strcpy(pathname,"/tmp") ;
    key = ftok(pathname,0x03);
    if(key==-1)
    {
        perror("ftok error");
        return -1;
    }
    printf("key=%d\n",key) ;
    shm_id=shmget(key,4096,IPC_CREAT|IPC_EXCL|0600);
    if(shm_id==-1)
    {
        perror("shmget error");
        return -1;
    }
    printf("shm_id=%d\n", shm_id) ;
    p_map=(people*)shmat(shm_id,NULL,0);
    memset(temp, 0x00, sizeof(temp)) ;
    strcpy(temp,"test") ;
    temp[4]='0';
    for(i = 0;i<3;i++)
    {
        temp[4]+=1;
        strncpy((p_map+i)->name,temp,5);
        (p_map+i)->age=0+i;
    }
    shmdt(p_map) ;
    return 0 ;
}

```

(2) 下列程序实现从共享内存读消息

shmread.c 源代码如下:

```

#include <stdio.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>
typedef struct{

```

```
char name[8];
int age;
} people;
int main(int argc, char** argv)
{
    int shm_id,i;
    key_t key;
    people *p_map;
    char pathname[30] ;

    strcpy(pathname,"/tmp") ;
    key = ftok(pathname,0x03);
    if(key == -1)
    {
        perror("ftok error");
        return -1;
    }
    printf("key=%d\n", key) ;
    shm_id = shmget(key,0, 0);
    if(shm_id == -1)
    {
        perror("shmget error");
        return -1;
    }
    printf("shm_id=%d\n", shm_id) ;
    p_map = (people*)shmat(shm_id,NULL,0);
    for(i = 0;i<3;i++)
    {
        printf( "name:%s\n",(*(p_map+i)).name );
        printf( "age %d\n",(*(p_map+i)).age );
    }
    if(shmdt(p_map) == -1)
    {
        perror("detach error");
        return -1;
    }
    return 0 ;
}
```

(3) 编译与执行

① 编译 gcc shmwrite.c -o shmwrite。

② 执行 ./shmwrite, 执行结果如下:

```
key=50453281
shm_id=688137
```

③ 编译 gcc shmread.c -o shmread。

④ 执行 ./shmread, 执行结果如下:

```
key=50453281
shm_id=688137
name:test1
age 0
```

```
name:test2  
age 1  
name:test3  
age 2
```

⑤ 再执行 `./shmwrite`，执行结果如下：

```
key=50453281  
shmget error: File exists
```

⑥ 使用 `ipcrm -m 688137` 删除此共享内存。

第 4 篇

Linux 文件

❖ 第 16 章 Linux 文件编程

学海聆听：

- 如果说我比别人看得更远些，那是因为我站在巨人的肩膀上。
- 归纳与演绎，内涵与外延。
- 三人行，必有我师；择其善者而从之，其不善者而改之。
- 学习型人生。
- 业精于勤，荒于嬉；行成于思，毁于随。
- 专注、专心、专业；人因专业而优秀，因专注而卓越。
- 自我规划，自我学习，自我约束，自我激励，自我管理。
- 学识来源于对外在的观察，对内心的思考。
- 善于发现的眼睛。
- 学而不厌，诲人不倦。
- 欲求木之长者，必固其本；欲求流之远者，必浚其源。
- 蓬生麻中，不扶而直；白沙在涅，与之俱黑。

第 16 章

Linux文件编程

文件系统是组织、存储、检索文件的系统。Linux 文件系统的表示和存储采用的是树形结构。对文件的编程一般包括两个部分：一是对文件和目录的管理，另一部分是对文件操作的编程。本章的文件系统函数部分主要介绍对文件和目录的管理编程，初级文件 I/O 函数章节介绍的是对文件操作的编程，本章第三部分介绍的是标准 I/O 的缓存和刷新，标准 I/O 函数的内容已在前面介绍，故不再重复。

16.1 文件系统函数

1. 文件属性

获取文件属性可通过 stat、fstat 和 lstat 函数完成，这三个函数的函数原型及其说明如下。

(1) stat、fstat 和 lstat 函数原型

所需头文件	<pre>#include <sys/types.h> #include <sys/stat.h> #include <unistd.h></pre>
函数说明	三个函数都用于获取文件属性，其中： stat: 根据文件路径得到文件属性 fstat: 根据文件描述符得到文件属性 lstat: 得到链接（link）文件本身的文件属性
函数原型	<pre>int stat(const char *file_name, struct stat *buf) int fstat(int filedes, struct stat *buf) int lstat(const char *file_name, struct stat *buf)</pre>
函数传入值	file_name: 文件名的全称 filedes: 文件描述符
函数传出值	buf: 文件信息结构
函数返回值	成功: 0 失败: -1, 错误原因存于 errno 中

续表

错误代码	ENOENT: 参数 file_name 指定的文件不存在 ENOTDIR: 路径中的目录存在但却非真正的目录 ELOOP: 欲打开的文件有过多的符号连接问题, 上限为 16 个符号连接 EFAULT: 参数 buf 为无效指针, 指向无法存在的内存空间 EACCESS: 存取文件时被拒绝 ENOMEM: 核心内存不足 ENAMETOOLONG: 参数 file_name 的路径名称太长
------	---

(2) struct stat 结构定义与说明

① struct stat 结构定义。

```
struct stat {  
    dev_t st_dev; /* 设备号 */  
    ino_t st_ino; /* inode 节点号 */  
    mode_t st_mode; /* 模式 */  
    nlink_t st_nlink; /* 硬连接 */  
    uid_t st_uid; /* 用户 ID */  
    gid_t st_gid; /* 组 ID */  
    dev_t st_rdev; /* 设备类型 */  
    off_t st_size; /* 文件字节数 */  
    unsigned long st_blksize; /* 块大小 */  
    unsigned long st_blocks; /* 块数 */  
    time_t st_atime; /* 最后一次访问时间 */  
    time_t st_mtime; /* 最后一次修改时间 */  
    time_t st_ctime; /* 最后一次改变时间(指属性) */  
};
```

② struct stat 各字段的具体说明如下。

字段名称	字段说明
st_dev	文件的设备编号
st_ino	文件的 i-node
st_mode	文件的类型和存取的权限
st_nlink	连到该文件的硬连接数目, 刚建立的文件值为 1
st_uid	文件所有者的用户识别码
st_gid	文件所有者的组织识别码
st_rdev	若此文件为装置设备文件, 则为其设备编号
st_size	文件大小, 以字节为单位
st_blksize	文件系统的 I/O
st_blocs	占用文件区块的个数, 每一区块大小为 512B
st_atime	文件最近一次被存取或被执行的时间, 一般只有在用 mknod、utime、read、write 与 tructate 时改变
st_mtime	文件最后一次被修改的时间, 一般只有在用 mknod、utime 和 write 时才会改变
st_ctime	i-node 最近一次被更改的时间, 此参数会在文件所有者、组、权限被更改时更新

③ struct stat 中 st_mode 位的意义。

宏	代表数字	意 义
S_IFMT	0170000	文件类型的位遮罩
S_IFSOCK	0140000	Socket
S_IFLNK	0120000	符号连接
S_IFREG	0100000	一般文件
S_IFBLK	0060000	区块装置
S_IFDIR	0040000	目录
S_IFCHR	0020000	字符装置
S_FIFO	0010000	先进先出
S_ISUID	04000	文件的 UID
S_ISGID	02000	文件的 GID
S_ISVTX	01000	文件的 sticky 位
S_IRUSR (S_IREAD)	00400	文件所有者具有可读取权限
S_IWUSR (S_IWRITE)	00200	文件所有者具有可写入权限
S_IXUSR (S_IEXEC)	00100	文件所有者具有可执行权限
S_IRGRP	00040	用户组具有可读取权限
S_IWGRP	00020	用户组具有可写入权限
S_IXGRP	00010	用户组具有可执行权限
S_IROTH	00004	其他用户具有可读取权限
S_IWOTH	00002	其他用户具有可写入权限
S_IXOTH	00001	其他用户具有可执行权限

④ 测试该文件类型。

利用下面的宏，可以根据 stat 结构的 st_mode 成员测试文件的类型。

宏	英文说明	中文说明
S_ISLNK(st_mode)	is it a symbolic link?	是否为符号链接
S_ISREG(st_mode)	regular file?	是否为一般文件
S_ISDIR(st_mode)	directory?	是否为目录
S_ISCHR(st_mode)	character device?	是否为字符设备
S_ISBLK(st_mode)	block device?	是否为块设备
S_ISFIFO(st_mode)	fifo?	是否为先进先出管道
S_ISSOCK(st_mode)	socket?	是否为 socket 描述符

(3) 文件属性函数举例

stat.c 源代码如下：

```
#include <sys/stat.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
int main()
{
    int fd ;
```



```
struct stat buf, buf1;
stat("/etc/passwd",&buf);
printf("stat: /etc/passwd file size = %d \n",buf.st_size);
printf("stat: /etc/passwd uid = %d \n",buf.st_uid);
printf("stat: /etc/passwd gid = %d \n",buf.st_gid);
printf("stat: /etc/passwd st_mode=0%o\n",buf.st_mode);
fd = open ("/etc/passwd",O_RDONLY);
fstat(fd,&buf1);
printf("fstat: /etc/passwd file size =%d\n ",buf1.st_size);
return 0 ;
}
```

编译 gcc stat.c -o stat。

执行 ./stat，执行结果如下：

```
stat: /etc/passwd file size = 2866
stat: /etc/passwd uid = 0
stat: /etc/passwd gid = 0
stat: /etc/passwd st_mode=0100644
fstat: /etc/passwd file size =2866
```

2. 目录读取函数

目录读取可通过 opendir、readdir 和 closedir 三个函数完成。这三个函数的函数原型及其说明如下。

(1) opendir、readdir、closedir 函数原型说明

opendir（打开目录）	
所需头文件	#include <sys/types.h> #include <dirent.h>
函数说明	opendir()用来打开参数 name 指定的目录，并返回 DIR*形态的目录流。与 open()类似，接下来对目录的读取和搜索都要使用此返回值
函数原型	DIR * opendir(const char * name)
函数传入值	name: 目录名称
函数返回值	成功: 返回 DIR*形态的目录流
	失败: NULL
错误代码	EACCESS: 权限不足 EMFILE: 已达到进程可同时打开的文件数上限 ENFILE: 已达到系统可同时打开的文件数上限 ENOTDIR: 参数 name 非真正的目录 ENOENT: 参数 name 指定的目录不存在，或是参数 name 为一空字符串 ENOMEM: 核心内存不足

readdir（读取目录）	
所需头文件	#include <sys/types.h> #include <dirent.h>

续表

readdir（读取目录）	
函数说明	readdir()返回参数 dir 目录流的下一个目录进入点。 结构 dirent 定义如下： struct dirent { ino_t d_ino; /*此目录进入点的 inode*/ off_t d_off; /*目录文件开头至此目录进入点的位移*/ unsigned short int d_reclen; /* d_name 的长度，不包含 NULL 字符*/ unsigned char d_type; /*d_name 所指的文件类型*/ char d_name[256]; /*文件名称*/ };
函数原型	struct dirent * readdir(DIR * dir)
函数传入值	DIR: opendir 打开的目录流
函数返回值	成功：返回下一个目录进入点
	失败：有错误发生或读取到目录文件尾则返回 NULL
错误代码	EBADF: 参数 dir 为无效的目录流

closedir（关闭目录）	
所需头文件	#include <sys/types.h> #include <dirent.h>
函数说明	closedir()关闭参数 dir 所指的目录流
函数原型	int closedir(DIR *dir)
函数传入值	DIR: opendir 打开的目录流
函数返回值	成功：0
	失败：-1，错误原因存于 errno 中
错误代码	EBADF: 参数 dir 为无效的目录流

(2) opendir、readdir、closedir 函数使用举例

opendir.c 源代码如下：

```
#include <sys/types.h>
#include <dirent.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
    DIR * dir;
    struct dirent * ptr;
    int i;
    dir =opendir("/etc/rc1.d");
    while((ptr = readdir(dir))!=NULL)
    {
        printf("d_name:%s\n",ptr->d_name);
    }
    closedir(dir);
    return 0 ;
}
```

编译 `gcc opendir.c -o opendir`。

执行 `./opendir`，执行结果如下：

```
d_name: K20vsftpd
d_name: K1lanacron
d_name: README
```

3. 文件和目录权限

(1) 文件访问许可说明

文件权限位在许多文件函数中都会用到，表 16-1 列出了文件访问权限宏及其含义。

表 16-1 文件访问权限表

宏	代表数字	说 明
S_IRWXU	00700	该文件所有者具有可读、可写及可执行的权限
S_IRUSR (S_IREAD)	00400	文件所有者具有可读取权限
S_IWUSR (S_IWRITE)	00200	文件所有者具有可写入权限
S_IXUSR (S_IEXEC)	00100	文件所有者具有可执行权限
S_IRWXG	00070	用户组具有可读、可写及可执行的权限
S_IRGRP	00040	用户组具有可读取权限
S_IWGRP	00020	用户组具有可写入权限
S_IXGRP	00010	用户组具有可执行权限
S_IRWXO	00007	其他用户具有可读、可写及可执行的权限
S_IROTH	00004	其他用户具有可读取权限
S_IWOTH	00002	其他用户具有可写入权限
S_IXOTH	00001	其他用户具有可执行权限

(2) 对目录中的可执行许可控制说明

利用全路径名称打开某个文件时，就涉及对该文件目录要有执行许可。

目录的读许可允许获得目录中的文件清单，可执行许可允许通过该目录访问文件。比如，要访问 `/etc/X10/XF80Config`，则必须拥有 `/、/etc、/etc/X10` 目录的可执行许可。

(3) access 函数原型

access（判断是否具有存取文件的权限）		
所需头文件	#include <unistd.h>	
函数说明	access() 会检查是否可以读/写某一已存在的文件	
函数原型	int access(const char * pathname,int mode)	
函数传入值	pathname: 文件路径全称	
	Mode（可为一种情况或几种情况的组合）	R_OK: 读
		W_OK: 写
		X_OK: 执行
		F_OK: 文件存在

续表

access（判断是否具有存取文件的权限）	
函数返回值	成功：0
	失败：-1，失败原因存在于 error 中
错误代码	EACCESS：参数 pathname 所指定的文件不符合所要求测试的权限 EROFS：欲测试写入权限的文件存在于只读文件系统内 EFAULT：参数 pathname 指针超出可存取内存空间 EINVAL：参数 mode 不正确 ENAMETOOLONG：参数 pathname 太长 ENOTDIR：参数 pathname 为一目录 ENOMEM：核心内存不足 ELOOP：参数 pathname 有过多的符号连接问题 EIO：I/O 存取错误
附加说明	使用 access() 作为用户认证方面的判断时要特别小心，例如，在 access() 后再做 open() 的空文件可能会造成系统安全上的问题

access 函数使用举例，access.c 源代码如下：

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    if (access("/etc/passwd", R_OK) ==0)
        printf("/etc/passwd can be read\n");
    return 0 ;
}
```

编译 gcc access.c -o access。

执行 ./access，执行结果如下：

```
/etc/passwd can be read
```

(4) umask 函数原型

umask（设置建立新文件时的权限遮罩）	
所需头文件	#include <sys/types.h> #include <sys/stat.h>
函数说明	umask() 会将 umask 值设成参数 mask&0777 后的值，然后将先前的 umask 值返回。在使用 open() 建立新文件时，该参数 mode 并非真正建立文件的权限，而是 (mode&~umask) 的权限值。例如，在建立文件时指定文件权限为 0666，通常 umask 值默认为 022，则该文件的真正权限为 0666&~022，即 0644
函数原型	mode_t umask(mode_t mask)
函数传入值	mask：设置的权限掩码
函数返回值	返回先前的 umask 值

(5) 改变文件或目录权限函数说明

改变文件或目录权限	
所需头文件	<code>#include <sys/types.h></code> <code>#include <unistd.h></code>
函数说明	依参数 mode 权限更改指定文件的权限。chmod 和 fchmod 的作用相同，只不过一个是通过文件全路径，另一个是通过文件描述符来修改文件的权限
函数原型	<code>int chmod(const char *path, mode_t mode)</code> <code>int fchmod(int fildes, mode_t mode)</code>
函数传入值	path: 文件全称
	mode: 文件权限位，见表 16-1
	fildes: 文件描述符
函数返回值	成功: 0
	失败: -1, 失败原因存在于 error 中

chmod、fchmod 函数使用举例，chmod.c 源代码如下：

```
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/types.h>
int main()
{
    int fd;
    chmod("/tmp/test.txt",S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
    fd = open ("/tmp/test1.txt",O_RDONLY);
    fchmod(fd,0331);
    close(fd);
    return 0 ;
}
```

- ① 编译 `gcc chmod.c -o chmod`。
- ② 在 /tmp 目录下建立 test.txt 和 test1.txt 两个文件。
- ③ 执行 ./chmod，使用 `ls -l /tmp/test*`，结果如下：

```
--wx-wx--x 1 zjkf db2iadml 0 2008-12-10 00:03 /tmp/test1.txt
-rw-r--r-- 1 zjkf db2iadml 0 2008-12-10 00:02 /tmp/test.txt
```

(6) 改变文件或目录所属用户和所属组函数说明

改变文件或目录权限	
所需头文件	<code>#include <sys/types.h></code> <code>#include <unistd.h></code>
函数说明	lchown 用来改变链接文件所属用户和组 chown 与 fchown 的作用相同，只不过一个是通过文件全路径，另一个是通过文件描述符来修改文件的所属用户和组
函数原型	<code>int chown(const char *path, uid_t owner, gid_t group)</code> <code>int fchown(int fd, uid_t owner, gid_t group)</code> <code>int lchown(const char *path, uid_t owner, gid_t group)</code>

续表

改变文件或目录权限	
函数传入值	path: 文件全称
	owner: 用户 ID 号
	group: 用户组 ID 号
函数返回值	成功: 0
	失败: -1, 失败原因存在于 error 中
附加说明	root 与文件所有者皆可改变文件组, 但所有者必须是参数 group 组的成员。当 root 用 chown() 改变文件所有者或组时, 该文件若具有 S_ISUID 或 S_ISGID 权限, 则会清除此权限位, 此外, 如果具有 S_ISGID 权限, 但不具有 S_IXGRP 位, 则该文件会被强制锁定, 文件模式会保留

4. 目录管理

目录是一种特殊文件, 其文件内容是该目录中的目录项。目录项内容包括索引节点编号、目录项名称长度以及名称本身。目录的常见操作有建立目录、删除目录、获取当前工作路径、改变当前工作路径等。

(1) 建立/删除目录函数说明

mkdir/rmdir (建立/删除目录)	
所需头文件	#include <sys/stat.h> #include <sys/types.h> #include <fcntl.h> #include <unistd.h>
函数说明	mkdir: 根据目录权限建立目录 rmdir: 删除指定的空目录
函数原型	int mkdir(const char *pathname, mode_t mode) int rmdir(const char *pathname)
函数传入值	pathname: 建立目录名称
	mode: 建立的目录权限, 见表 16-1
函数返回值	成功: 0
	失败: -1, 失败原因存于 error 中

mkdir、rmdir 函数使用举例, mkdir.c 源代码如下:

```
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
    int flag ;
    flag = mkdir("./path1", 0730);
    if ( 0 != flag )
    {
        perror("mkdir error") ;
        return -1 ;
    }
}
```

```
    }
    flag = rmdir("./path2");
    if ( 0 != flag )
    {
        perror("rmdir error") ;
        return -1 ;
    }
    return 0 ;
}
```

- ① 编译 gcc mkdir.c -o mkdir。
- ② 使用 mkdir path2 建立一个新目录。
- ③ 执行 ./mkdir，执行结果为：在当前目录下新建了 path1 目录，删除了 path2 目录。

(2) 改变/获取当前工作目录函数说明

chdir（改变当前的工作目录）	
所需头文件	#include <unistd.h>
函数说明	chdir()用来将当前的工作目录改变成以参数 path 所指的目录
函数原型	int chdir(const char * path)
函数传入值	path: 到达的目录
函数返回值	成功: 0
	失败: -1, 失败原因存于 error 中

fchdir（改变当前的工作目录）	
所需头文件	#include <unistd.h>
函数说明	fchdir()用来将当前的工作目录改变成以参数 fd 所指的文件描述符
函数原型	int fchdir(int fd)
函数传入值	文件描述符
函数返回值	成功: 0
	失败: -1, 失败原因存于 error 中

getcwd（取得当前的工作目录）	
所需头文件	#include <unistd.h>
函数说明	getcwd()会将当前的工作目录绝对路径复制到参数 buf 所指的内存空间，参数 size 为 buf 的空间大小。在调用此函数时，buf 所指的内存空间要足够大，若工作目录绝对路径的字符串长度超过参数 size 大小，则返回值为 NULL，errno 的值则为 ERANGE。若参数 buf 为 NULL，getcwd()会依参数 size 的大小自动配置内存（使用 malloc()），如果参数 size 也为 0，则 getcwd()会依工作目录绝对路径的字符串长度来决定所配置的内存大小，进程可以在使用完此字符串后利用 free(buf)释放此空间
函数原型	char *getcwd(char *buf, size_t size)
函数传入值	buf: 存放当前工作目录的内存地址
	size: buf 空间大小
函数返回值	成功: 将结果复制到参数 buf 所指的内存空间，或是返回自动配置的字符串指针
	失败: NULL, 失败原因存在于 error 中

chdir、fchdir、getcwd 函数使用举例，chdir.c 源代码如下：

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int main()
{
    int flag;
    int fd;
    char path[128];
    flag=chdir("/tmp");
    if ( 0 != flag )
    {
        perror("chdir error") ;
        return -1 ;
    }
    printf("current working directory: %s\n",getcwd(path, 127));
    fd = open("/usr/bin",O_RDONLY);
    flag=fchdir(fd);
    if ( 0 != flag )
    {
        perror("chdir error") ;
        return -1 ;
    }
    printf("current working directory : %s \n",getcwd(path, 127));
    close(fd);
    return 0 ;
}
```

编译 gcc chdir.c -o chdir。

执行 ./chdir，执行结果如下：

```
current working directory: /tmp
current working directory : /usr/bin
```

5. 文件管理

(1) 硬链接与符号链接说明

硬链接是指同一文件系统中，有多个文件指向文件系统中的同一个 inode 节点。使用 link 函数可建立硬链接。

符号链接又称为软链接，它指向的是目的文件名，其作用类似于 windows 的快捷方式。符号可以跨文件系统，而硬链接不能。

(2) 改变文件大小函数说明

truncate/ftruncate（截短文件）	
所需头文件	#include <unistd.h>
函数说明	将文件大小改为参数 length 指定的大小。如果原来的文件比参数 length 大，则超过的部分会被删去
函数比较	两个函数的功能相同。只不过一个是通过文件全路径，一个是通过文件描述符达到目的。类似作用的函数都是在另一个函数前加 f

续表

truncate/ftruncate（截短文件）	
函数原型	<code>int truncate(const char *path, off_t length)</code> <code>int ftruncate(int fd, off_t length)</code>
函数传入值	<code>path</code> : 文件全称
	<code>length</code> : 截断文件长度
	<code>fd</code> : 文件描述符
函数返回值	成功: 0
	失败: -1, 失败原因存于 <code>error</code> 中

（3）建立硬链接、删除文件函数说明

link（建立硬链接）	
所需头文件	<code>#include <unistd.h></code>
函数说明	<code>link()</code> 以参数 <code>newpath</code> 指定的名称来建立一个新的链接（硬链接）到参数 <code>oldpath</code> 所指定的已存在的文件。如果参数 <code>newpath</code> 指定的名称为一个已存在的文件，则不会建立连接
函数原型	<code>int link(const char *oldpath, const char * newpath)</code>
函数传入值	<code>oldpath</code> : 原文件名称
	<code>newpath</code> : 新链接的名称
函数返回值	成功: 0
	失败: -1, 失败原因存于 <code>error</code> 中
附加说明	<code>link()</code> 所建立的硬链接无法跨越不同的文件系统，如需要，请改用 <code>symlink()</code>

unlink（删除文件）	
所需头文件	<code>#include <unistd.h></code>
函数说明	<code>unlink()</code> 会删除参数 <code>pathname</code> 指定的文件。如果该文件名为最后的链接点，但有其他进程打开了此文件，则在所有关于此文件的文件描述符皆关闭后才会删除。如果参数 <code>pathname</code> 为一符号链接，则此链接会被删除
函数原型	<code>int unlink(const char * pathname)</code>
函数传入值	<code>pathname</code> : 文件全路径名称
函数返回值	成功: 0
	失败: -1, 失败原因存于 <code>error</code> 中
unlink 成功的先决条件	① 作用于非目录文件 ② 对包含该目录项的目录必须具有写和执行许可 ③ 若目录具有黏着位，则必须满足：拥有文件，拥有目录或具有超级用户特权

link、unlink 函数使用举例，link.c 源代码如下：

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    int flag ;
    flag = link("oldfile","filelink");
    if ( 0 != flag )
```

```
{
    perror("link error") ;
    return -1 ;
}
flag = unlink("test.txt");
if ( 0 != flag )
{
    perror("unlink error") ;
    return -1 ;
}
return 0 ;
}
```

- ① 编译 gcc link.c -o link。
- ② 在当前目录用>oldfile、>test.txt 创建两个文件。
- ③ 用 vi 打开 oldfile 文件并增加内容。
- ④ 执行 ./link 后，发现 test.txt 被删除，而 oldfile 和 filelink 两个文件的内容一致。

(4) 建立符号链接/读取符号链接函数说明

symlink（建立文件符号链接）	
所需头文件	#include <unistd.h>
函数说明	symlink()以参数 newpath 指定的名称来建立一个新的链接（符号链接）到参数 oldpath 所指定的已存在的文件。参数 oldpath 指定的文件一定要存在。如果参数 newpath 指定的名称为一个已存在的文件，则不会建立链接
函数原型	int symlink(const char *oldpath,const char *newpath)
函数传入值	oldpath: 原文件或原目录
	newpath: 建立符合链接的文件和目录
函数返回值	成功: 0
	失败: -1, 失败原因存于 error 中

readlink（取得符号链接所指的文件）	
所需头文件	#include <unistd.h>
函数说明	readlink()会将参数 path 的符号链接内容存到参数 buf 所指的内存空间，返回的内容不是以 NULL 为字符串结尾，但会将字符串的字符数返回。若参数 bufsiz 小于符号链接的内容长度，过长的内容会被截断
函数原型	int readlink(const char *path ,char *buf,size_t bufsiz)
函数传入值	path: 符号链接全路径名称
	bufsiz: 最大读取字节数
函数传出值	buf: 读取符号链接内容
函数返回值	成功: 成功读取的字节数
	失败: -1, 失败原因存于 error 中

symlink、readlink 函数使用举例，symlink.c 源代码如下：

```
#include <stdio.h>
#include <unistd.h>
```

```
int main()
{
    int flag ;
    int length ;
    char buf[512] ;
    flag=symlink("/etc/passwd","pass");
    if ( 0 != flag )
    {
        perror("symlink error") ;
        return -1 ;
    }
    length=readlink("pass" , buf, (512-1));
    if ( length < 0 )
    {
        perror("readlink error") ;
        return -1 ;
    }
    printf("buf=%s\n",buf) ;
    return 0 ;
}
```

① 编译 gcc symlink.c -o symlink。

② 执行 ./symlink，执行结果如下：

```
buf=/etc/passwd
```

③ 执行 l pass，可以看出此文件是一个链接文件。

```
lrwxrwxrwx 1 zjkg db2iadml 11 2008-12-10 18:05 pass -> /etc/passwd
```

(5) 删除文件、重命名文件函数说明

remove（删除文件）	
所需头文件	#include <stdio.h>
函数说明	remove()会删除参数 pathname 指定的文件。如果参数 pathname 为一个文件，则调用 unlink()处理；若参数 pathname 为一个目录，则调用 rmdir()来处理。请参考 unlink()与 rmdir()
函数原型	int remove(const char * pathname)
函数传入值	pathname: 删除的文件名称或目录名称
函数返回值	成功: 0
	失败: -1, 失败原因存于 error 中
附加说明	remove 可删除文件和目录，分别等同于 unlink 和 rmdir

rename（更改文件名称或位置）	
所需头文件	#include <stdio.h>
函数说明	rename()会将参数 oldpath 所指定的文件名称改为参数 newpath 所指的文件名称。若 newpath 所指定的文件已存在，则会被删除
函数原型	int rename(const char *oldpath,const char *newpath)
函数传入值	oldpath: 原文件名称
	newpath: 新文件名称
函数返回值	成功: 0
	失败: -1, 失败原因存于 error 中

rename 函数使用举例，rename.c 源代码如下：

```
#include <stdio.h>
int main(int argc, char **argv)
{
    if(argc<3){
        printf("Usage: %s old_name new_name\n",argv[0]);
        return -1;
    }
    printf("%s=>%s\n", argv[1], argv[2]);
    if(rename(argv[1], argv[2])<0)
    {
        perror("error!");
        return -1;
    }
    else
        printf("ok!\n");
    return 0 ;
}
```

- ① 编译 gcc rename.c -o rename。
- ② 用>aa 新建一个文件，执行./rename aa aaa 发现文件 aa 更名为 aaa。

6. dup 和 dup2 函数

dup 和 dup2 都可用来复制一个已存在的文件描述符，使两个文件描述符指向同一个 file 结构体。如果两个文件描述符指向同一个 file 结构体，文件状态标志和读写位置只保存一份在 file 结构体中，并且 file 结构体的引用计数是 2。dup 与 dup2 的函数原型说明如下。

dup/dup2（复制文件描述符）	
所需头文件	#include <unistd.h>
函数说明	dup: 复制 oldfd 文件描述符，返回该进程未使用的最小文件描述符 dup2: 复制 oldfd 文件描述符，返回 newfd 文件描述符
函数原型	int dup(int oldfd) int dup2(int oldfd, int newfd)
函数传入值	oldfd: 复制的源文件描述符
	newfd: 指定新的文件描述符
函数返回值	成功: 返回新的文件描述符
	失败: -1, 失败原因存于 error 中
附加说明	dup2 可以用 newfd 参数指定新文件描述符的数值。如果 newfd 当前已经打开，则先将其关闭，再做 dup2 操作，如果 oldfd 等于 newfd，则 dup2 直接返回 newfd，而不用先关闭 newfd 再复制

下面的 dup.c 程序是 dup 和 dup2 函数的用法演示程序。图 16-1 演示了 dup.c 程序中文件描述符变化图。

dup.c 源代码如下：

```
#include <unistd.h>
#include <sys/stat.h>
```

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    int fd, save_fd;
    char msg[] = "This is a test\n";
    fd = open("somefile", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
    if(fd<0) {
        perror("open");
        exit(1);
    }
    save_fd = dup(STDOUT_FILENO);
    dup2(fd, STDOUT_FILENO);
    close(fd);
    write(STDOUT_FILENO, msg, strlen(msg));
    dup2(save_fd, STDOUT_FILENO);
    write(STDOUT_FILENO, msg, strlen(msg));
    close(save_fd);
    return 0;
}
```

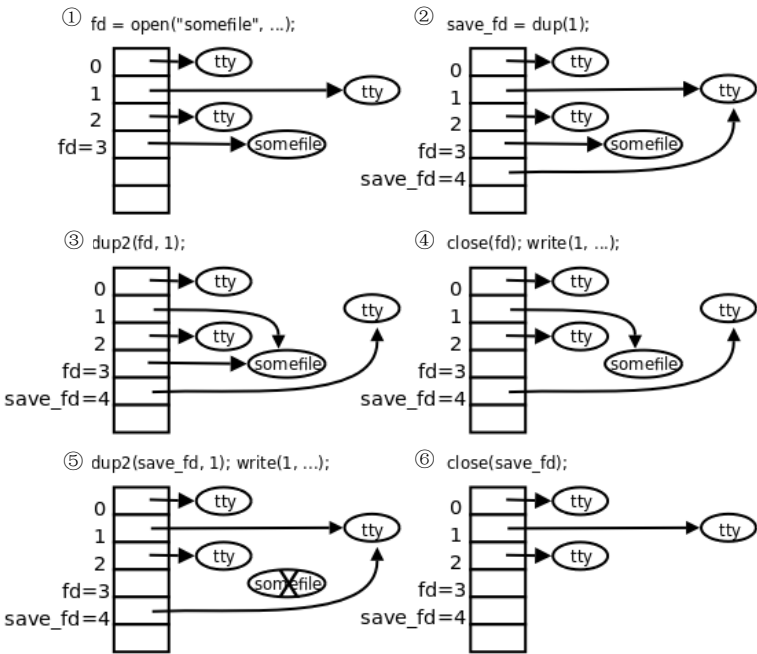


图 16-1 dup 函数实现原理图

编译 `gcc dup.c -o dup。`

执行 `./dup`，执行结果如下：

This is a test

16.2 初级文件 I/O 函数

初级文件 I/O 函数是最初 UNIX 系统实现的文件 I/O 函数。现在文件编程都使用带缓存的文件 I/O 函数（即标准 I/O 函数，也是以 f 开头的函数）。但在应用编程中，会遇到前辈们遗留下来的大量的代码，许多是使用的初级文件 I/O 函数，维护和阅读这些代码需要知晓初级文件 I/O 函数，同时对文件 I/O 进行一些特别的操作和控制时一般也要使用到初级文件 I/O 函数。

1. 初级文件 I/O 函数原型

本节主要介绍初级文件 I/O 操作函数，初级 I/O 函数常被称为不用缓存的 I/O 函数。初级文件 I/O 函数主要包括 open、read、write、create、lseek 和 close。这六个函数的函数原型及说明如下。

open（打开文件）	
所需头文件	<pre>#include <sys/types.h> #include <sys/stat.h> #include <fcntl.h></pre>
函数说明	根据路径名打开文件
函数原型	<pre>int open(const char *pathname, int flags) int open(const char *pathname, int flags, mode_t mode)</pre>
函数传入参数	pathname: 打开文件名全称
	flags 的参数意义如下： ① O_RDONLY: 以只读方式打开文件 ② O_WRONLY: 以只写方式打开文件 ③ O_RDWR: 以可读写方式打开文件 上述三种旗标是互斥的，即不可同时使用，但可与下列旗标利用 OR () 运算符组合。 ④ O_CREAT: 若欲打开的文件不存在，则自动建立该文件 ⑤ O_EXCL: 如果 O_CREAT 也被设置，此指令会去检查文件是否存在。若文件不存在，则建立该文件，否则将导致打开文件错误。此外，若 O_CREAT 与 O_EXCL 同时设置，并且欲打开的文件为符号链接，则会打开文件失败 ⑥ O_NOCTTY: 如果欲打开的文件为终端机设备，则不会将该终端机当成进程控制终端机 ⑦ O_TRUNC: 若文件存在并且以可写的方式打开，此旗标会令文件长度清为 0，而原来存于该文件的资料也会消失 ⑧ O_APPEND: 当读写文件时会从文件尾开始移动，也就是所写入的数据会以附加的方式加入文件后面 ⑨ O_NONBLOCK: 以不可阻断的方式打开文件，也就是无论有无数据读取或等待，都会立即返回进程之中 ⑩ O_NDELAY: 同 O_NONBLOCK ⑪ O_SYNC: 以同步方式打开文件 ⑫ O_NOFOLLOW: 如果参数 pathname 所指的文件为一个符号链接，会令打开文件失败 ⑬ O_DIRECTORY: 如果参数 pathname 所指的文件并非为一个目录，会令打开文件失败
	Mode: 被打开文件的存取权限，含义同表 16-1

续表

open（打开文件）	
函数返回值	成功：文件描述符
	失败：-1，错误代码存放在 error 中
错误代码	EEXIST：参数 pathname 所指的文件已存在，却使用了 O_CREAT 和 O_EXCL 旗标 EACCESS：参数 pathname 所指的文件不符合所要求测试的权限 EROFS：欲测试写入权限的文件存在于只读文件系统内 EFAULT：参数 pathname 指针超出可存取的内存空间 EINVAL：参数 mode 不正确 ENAMETOOLONG：参数 pathname 太长 ENOTDIR：参数 pathname 不是目录 ENOMEM：核心内存不足 ELOOP：参数 pathname 有过多符号连接问题 EIO：I/O 存取错误
附加说明	使用 access() 做用户认证方面的判断时要特别小心，例如，在 access() 后再做 open() 空文件可能会造成系统安全上的问题

close（关闭文件）	
所需头文件	#include <unistd.h>
函数说明	当使用完文件后，若已不再需要，则可使用 close() 关闭该文件，close() 会让数据写回磁盘，并释放该文件所占用的资源。参数 fd 为先前由 open() 或 creat() 所返回的文件描述符
函数原型	int close(int fd)
函数传入参数	fd：文件描述符
函数返回值	成功：0
	失败：-1，错误代码存放在 error 中
错误代码	EBADF：参数 fd 非有效的文件描述符或该文件已关闭
附加说明	虽然在进程结束时，系统会自动关闭已打开的文件，但仍建议自行关闭文件，并确实检查返回值

creat（建立文件）	
所需头文件	#include <sys/types.h> #include <sys/stat.h> #include <fcntl.h>
函数说明	参数 pathname 指向欲建立的文件路径字符串。creat() 相当于使用下列 open() 调用，open(const char * pathname , (O_CREAT O_WRONLY O_TRUNC))
函数原型	int creat(const char * pathname, mode_t mode)
函数传入参数	pathname：打开文件名全称
	mode：参看 open 中的解释
函数返回值	成功：文件描述符
	失败：-1，错误代码存放在 error 中
错误代码	EBADF：参数 fd 非有效的文件描述符或该文件已关闭
附加说明	虽然在进程结束时，系统会自动关闭已打开的文件，但仍建议自行关闭文件，并确实检查返回值

续表

read（由已打开的文件读取数据）	
所需头文件	#include <unistd.h>
函数说明	read()会把参数 fd 所指的文件传送 count 个字节到 buf 指针所指的内存中。若参数 count 为 0，则 read()不会起作用并返回 0，返回值为实际读取到的字节数，如果返回 0，表示已到达文件尾或是无可读取的数据，此外，文件读写位置会随读取到的字节移动
函数原型	ssize_t read(int fd, void *buf, size_t count)
函数传入参数	Fd: 文件描述符
	count: 最大读取字节数
函数传出参数	buf: 读取数据的首地址
函数返回值	成功: 实际读取字节数
	失败: -1, 错误代码存放在 error 中
错误代码	EINTR: 此调用被信号所中断
	EAGAIN: 当使用不可阻断 I/O 时 (O_NONBLOCK)，若无数据可读取，则返回此值
	EBADF: 参数 fd 非有效的文件描述符，或该文件已关闭
附加说明	如果顺利，read()会返回实际读到的字节数，最好能将返回值与参数 count 进行比较，若返回的字节数比要求读取的字节数少，则有可能读到了文件尾、从管道 (pipe) 或终端机读取，或者是 read()被信号中断了读取动作。当有错误发生时，则返回-1，错误代码存入 errno 中，而文件读写位置则无法预期

write（将数据写入已打开的文件内）	
所需头文件	#include <unistd.h>
函数说明	write()会把参数 buf 所指的内存写入 count 个字节到参数 fd 所指的文件内。当然，文件读写位置也会随之移动
函数原型	ssize_t write (int fd, const void *buf, size_t count)
函数传入参数	Fd: 文件描述符
	buf: 写入数据的首地址
	count: 最大写入的字节数
函数返回值	成功: 实际写入的字节数
	失败: -1, 错误代码存放在 error 中
错误代码	EINTR: 此调用被信号所中断
	EAGAIN: 当使用不可阻断 I/O 时 (O_NONBLOCK)，若无数据可读取，则返回此值
	EADF: 参数 fd 非有效的文件描述符，或该文件已关闭

lseek（移动文件流的读写位置）	
所需头文件	#include <sys/types.h> #include <unistd.h>
函数说明	每一个已打开的文件都有一个读写位置，当打开文件时，通常其读写位置是指向文件开头，若是以附加的方式打开文件（如 O_APPEND），则读写位置会指向文件尾。当调用 read()或 write()函数时，读写位置会随之增加，lseek()可用来改变该文件的读写位置
函数原型	off_t lseek(int fildes, off_t offset, int whence)

续表

lseek（移动文件流的读写位置）			
函数传入值	Fildes: 已打开的文件描述符		
	offset: 根据参数 whence 来移动读写位置的位移数。可为正、负、0		
	whence（既可以用宏，也可以用数字）		
	起始点	宏表示符号	数字表示
	文件首	SEEK_SET	0
	当前位置	SEEK_CUR	1
	文件末尾	SEEK_END	2
函数返回值	成功：目前的读写位置，也就是距离文件开头多少个字节		
	错误：-1, errno 会存放错误代码		
常见使用方法	① 欲将读写位置移到文件开头时用 lseek(fildes,0,SEEK_SET)		
	② 欲将读写位置移到文件尾时用 lseek(fildes,0,SEEK_END)		
	③ 想要取得目前文件位置时用 lseek(fildes,0,SEEK_CUR)		
附加说明	Linux 系统不允许 lseek（）对 tty 起作用，此项动作会令 lseek（）返回 ESPIPE		

2. 初级文件 I/O 函数使用示例

该示例程序首先打开一个已创建的文件，然后对此文件进行读写操作。接着，写入“Hello good morning, nice to meet you, what is your name?”，随后使用 lseek 函数将文件指针移到文件开始处，并读出 10 个字节，最后将其打印出来。

write.c 源代码如下：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
int main(void)
{
    int i,fd,size,len;
    char *buf="Hello good morning, nice to meet you, what is your name?";
    char buf_r[10+1];
    len = strlen(buf);
    /*首先调用 open 函数，并指定相应的文件权限*/
    if((fd = open("/tmp/hello.txt", O_CREAT | O_TRUNC | O_RDWR, 0660 ))<0){
        perror("open:");
        return -1 ;
    }
    else
        printf("open file:hello.txt %d\n",fd);
    /*调用 write 函数，将 buf 中的内容写入打开的文件中*/
    if((size = write( fd, buf, len)) < 0){
        perror("write:");
        return -1 ;
    }
    else
        printf("write:%s\n",buf);
}
```

```
/*调用 lseek 函数将文件指针移到文件起始位置，并读取文件中的 10 个字节*/
memset(buf_r, 0x00, sizeof(buf_r)) ;
lseek( fd, 0, SEEK_SET );
if((size = read( fd, buf_r, 10))<0){
    perror("read:");
    return -1 ;
}
else
    printf("read form file:%s\n",buf_r);
if( close(fd) < 0 ){
    perror("close:");
    return -1 ;
}
else
    printf("close hello.txt\n");
return -1 ;
}
```

编译 gcc write.c -o write

执行./write，执行结果如下：

```
open file:hello.txt 3
write:Hello good morning, nice to meet you, what is your name?
read form file:Hello good
close hello.txt
```

3. fcntl 函数

(1) fcntl 函数说明

Linux 通常采用给文件上锁的方法来避免共享资源产生竞争的状态。

文件锁包括建议性锁和强制性锁。建议性锁要求每个上锁文件的进程都要检查是否有锁存在，并且尊重已有的锁。在一般情况下，内核和系统都不使用建议性锁。强制性锁是由内核执行的锁，当一个文件被上锁进行写入操作的时候，内核将阻止其他任何进程对其进行读写操作。采用强制性锁对性能的影响很大，每次读写操作前都必须检查是否有锁存在。

在 Linux 中，fcntl 函数不仅可以给文件施加建议性锁，还可以施加强制性锁。同时，fcntl 函数还能对文件的某一条记录进行上锁（称为记录锁）。记录锁又可分为读取锁和写入锁，其中读取锁又称为共享锁，多个进程都能在文件的同一部分建立读取锁。写入锁又称为排斥锁，在任何时刻只能有一个进程在文件的同一部分建立写入锁。注意：在文件的同一部分不能同时建立读取锁和写入锁。

(2) fcntl 函数原型

fcntl（文件描述符操作）	
所需头文件	#include <unistd.h> #include <fcntl.h>
函数说明	fcntl()用来操作文件描述符的一些特性。参数 fd 代表欲设置的文件描述符，参数 cmd 代表欲操作的指令

续表

fcntl（文件描述符操作）	
函数原型	<pre>int fcntl(int fd, int cmd) int fcntl(int fd, int cmd, long arg) int fcntl(int fd, int cmd, struct flock * lock)</pre>
函数传入参数	<p>Fd: 文件描述符</p> <p>cmd 的几种情况如下。</p> <p>① F_DUPFD: 用来查找大于或等于参数 arg 的最小且仍未使用的文件描述符，并且复制参数 fd 的文件描述符，执行成功则返回新复制的文件描述符。请参考 dup2()。F_GETFD 取得 close-on-exec 旗标，若此旗标的 FD_CLOEXEC 位为 0，代表在调用 exec() 相关函数时，文件将不会关闭</p> <p>② F_SETFD: 设置 close-on-exec 旗标。该旗标由参数 arg 的 FD_CLOEXEC 位决定</p> <p>③ F_GETFL: 取得文件描述符状态旗标，此旗标为 open（）的参数 flags</p> <p>④ F_SETFL: 设置文件描述符状态旗标，参数 arg 为新旗标，但只允许 O_APPEND、O_NONBLOCK 和 O_ASYNC 位的改变，其他位的改变将不受影响</p> <p>⑤ F_GETLK: 取得文件锁定的状态</p> <p>⑥ F_SETLK: 设置文件锁定的状态。此时 flock 结构的 l_type 值必须是 F_RDLCK、F_WRLCK 或 F_UNLCK。如果无法建立锁定，则返回-1，错误代码为 EACCES 或 EAGAIN</p> <p>⑦ F_SETLKW: 与 F_SETLK 的作用相同，但是无法建立锁定时，此调用会一直等到锁定动作成功为止。若在等待锁定的过程中被信号中断，则会立即返回-1，错误代码为 EINTR</p>
	<p>lock: 锁操作，具体说明见表 16-2</p>
	<p>成功: 0</p>
	<p>失败: -1，错误代码存放在 error 中</p>

表 16-2 列出了 struct flock 结构定义。

表 16-2 struct flock 结构定义表

lock 结构变量取值	
struct flock 结构定义	<pre>struct flock{ short l_type; off_t l_start; short l_whence; off_t l_len; pid_t l_pid; }</pre>
l_type	F_RDLCK: 读取锁（共享锁）
	F_WRLCK: 写入锁（排斥锁）
	F_UNLCK: 解锁
l_start	相对位移量（字节）
l_whence: 相对位移量的起点（同 lseek 的 whence）	SEEK_SET: 当前位置为文件的开头，新位置为偏移量的大小
	SEEK_CUR: 当前位置为文件指针的位置，新位置为当前位置加上偏移量
	SEEK_END: 当前位置为文件的结尾，新位置为文件的大小加上偏移量的大小
l_len	加锁区域的长度
l_pid	加锁的进程 ID
对整个文件加锁的技巧	通常的方法是将 l_start 设置为 0、l_whence 设置为 SEEK_SET、l_len 说明为 0

(3) fcntl 常用的用途

- ① 复制描述符 (cmd=F_DUPFD)。
- ② 获取/设置文件描述符标志 (cmd=F_GETFD or F_SETFD)。
- ③ 获取/设置文件状态标志 (cmd=F_GETFL or F_SETFL)。
- ④ 获取/设置记录锁 (cmd=F_GETLK、F_SETLK or F_SETLKW)。
- ⑤ 获取/设置异步 I/O (cmd=F_GETOWN、F_SETOWN、F_GETSIG or F_SETSIG)。

(4) fcntl 使用示例

下面的实例是测试文件的写锁。fcntl_write.c 源代码如下:

```
#include <unistd.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
void lock_set(int fd, int type)
{
    struct flock lock;
    lock.l_whence = SEEK_SET; /*赋值 lock 结构体*/
    lock.l_start = 0;
    lock.l_len = 0;
    while(1){
        lock.l_type = type;
        /*根据不同的 type 值给文件上锁或解锁*/
        if((fcntl(fd, F_SETLK, &lock)) == 0){
            if( lock.l_type == F_RDLCK )
                printf("read lock set by %d\n",getpid());
            else if( lock.l_type == F_WRLCK )
                printf("write lock set by %d\n",getpid());
            else if( lock.l_type == F_UNLCK )
                printf("release lock by %d\n",getpid());
            return;
        }
        /*判断文件是否可以上锁*/
        fcntl(fd, F_GETLK,&lock);
        /*判断文件不能上锁的原因*/
        if(lock.l_type != F_UNLCK){
            /*该文件已有写入锁*/
            if( lock.l_type == F_RDLCK )
                printf("read lock already set by %d\n",lock.l_pid);
            /*该文件已有读取锁*/
            else if( lock.l_type == F_WRLCK )
                printf("write lock already set by %d\n",lock.l_pid);
            getchar();
        }
    }
}
```

```
int main(void)
{
    int fd;
    fd=open("why.txt",O_RDWR | O_CREAT, 0600);
    if(fd < 0){
        perror("open");
        return -1;
    }
    /*给文件上加写入锁*/
    lock_set(fd, F_WRLCK);
    getchar();
    /*给文件解锁*/
    lock_set(fd, F_UNLCK);
    getchar();
    close(fd);
    return 0;
}
```

① 编译 `gcc fcntl_write.c -o fcntl_write`。

② 在终端一上执行 `./fcntl_write`，执行结果如下：

```
write lock set by 11463
```

③ 在终端二上执行 `./fcntl_write`，执行结果如下：

```
write lock already set by 11463
```

④ 在终端一上按回车键，执行结果如下：

```
release lock by 11463
```

⑤ 在终端二上按两次回车键，执行结果如下：

```
write lock set by 11466
release lock by 11466
```

⑥ 由此可见，写入锁为互斥锁，即同一时刻只能有一个写入锁存在。

16.3 标准 I/O 的缓冲和刷新

1. I/O 缓冲说明

标准 I/O 提供缓冲区（又称缓存）的目的是尽可能减少使用 `read` 和 `write` 调用的数量。它也对每个 I/O 流自动地进行缓冲区管理，避免了应用程序需要考虑这一点所带来的麻烦。

C 标准库的 I/O 缓冲区有三种类型：全缓冲、行缓冲和无缓冲。当用户程序调用库函数进行写操作时，三种不同类型的缓冲区具有如下不同的特性。

① 全缓冲：如果缓冲区写满了，就写回内核。常规文件通常是全缓冲的。

② 行缓冲：如果用户程序写的数据中有换行符，就把这一行写回内核，或者如果缓冲区写满

了，就写回内核。标准输入和标准输出对应终端设备时通常是行缓冲的。

③ 无缓冲：用户程序每次调库函数进行写操作都要通过系统调用写回内核。标准错误输出通常是无缓冲的，这样用户程序产生的错误信息可以尽快输出到设备。

2. 刷新缓存

有时，数据在写入设备之前是先存放在缓冲区中的，此时可利用 `fflush` 函数把缓冲区中的数据强制刷新到设备上。

(1) fflush 函数原型

fflush（刷新缓冲区中的数据）	
所需头文件	#include <stdio.h>
函数说明	强迫将缓冲区内的数据写回参数 stream 指定的文件中。如果参数 stream 为 NULL, fflush() 会将所有打开的文件数据更新
函数原型	int fflush(FILE* stream)
函数传入值	stream: 已打开的文件指针
函数返回值	成功: 0
	错误: 返回 EOF, 错误代码存于 errno 中
错误代码	EBADF: 参数 stream 指定的文件未被打开, 或打开状态为只读

(2) fflush 函数使用示例

fflush.c 源代码如下：

```
#include <stdio.h>
int main()
{
    printf("hello world");
    /*比较去掉 fflush 函数和加上此函数两种执行结果的不同*/
    fflush(stdout);
    sleep(30) ;
    return 0 ;
}
```

① 编译 `gcc fflush.c -o fflush`。

② 执行 `./fflush`，执行结果如下：

```
hello world
```

③ 在上述程序中，若包含 `fflush` 函数语句，则输出字符会立刻打印到屏幕上；若不包含，则要过 30 秒钟才打印到屏幕上，即在程序退出后才打印到屏幕上。

第 5 篇

网络编程

❖ 第 17 章 网络知识基础

❖ 第 18 章 Socket 编程

学海聆听：

- 非淡泊无以明志，非宁静无以致远。
- 他山之石，可以攻玉。
- 文章千古事，得失寸草心。
- 自信、自律、自省、自励、自珍、自爱、自知、自重、自尊、自立、自强、自现。
- 敢于梦想，追求梦想，实现梦想，超越梦想。
- 世间万事万物各有各的规律，各有各的办法与学问。
- 知道自己的目标，列出目标所需的知识点，寻找合适的方法，制定详细的计划，持之以恒地努力。
- 授人以鱼，不如授人以渔。
- 苦学与巧学，天赋与勤奋。
- 把事情做好既需要知识，也需要态度、吃苦精神、毅力和意志。
- 法制精神，科技理性，人文底蕴。
- 本体论，认识论，方法论。是什么，为什么，怎么样。

第 17 章

网络知识基础

计算机网络就是通过一定形式连接起来的一组计算机系统，它需要四个要素的支持，即通信线路和通信设备、有独立功能的计算机、网络软件、能实现数据通信与资源共享。计算机网络具有两大参考模型，分别为 OSI 参考模型和 TCP/IP 参考模型，其中 OSI 参考模型为理论模型，TCP/IP 参考模型则已成为互联网事实的工业标准，现在的通信网络一般都采用 TCP/IP 协议簇，而应用编程则都采用 Socket 套接字进行编程。

17.1 网络体系结构及协议

17.1.1 网络体系结构概念

1. 网络体系结构的重要概念

网络体系结构涉及以下几个重要的概念。

① 协议：为计算机网络中的数据交换而建立的规则、标准或约定的集合。

② 通信协议：通信双方必须共同遵守的规则和约定称为通信协议。通信双方对数据的理解需要建立在约定与协议之上。

③ 接口：相邻两层之间的边界，在接口处规定了低层向上层提供的服务以及服务所使用的形式规范语句（服务原语）。

④ 服务：某一层提供的功能，并能通过接口提供给她相邻的上层。

⑤ 网络体系结构：对计算机网络的各层功能的精确定义及其各层遵守协议的集合。

⑥ 协议栈：网络各层协议按层次顺序排列而成的协议序列。

⑦ 点到点：体现在物理上的两两连接，是物理拓扑结构，如光纤就必须是点到点的连接。点到点协议体现在 IP 网络层或以下两层。IP 网络层是两两路由器进行点到点通信，中间没有跨越其他通信设备。点到点传输的优点是发送端设备送出数据后，它的任务已经完成，不需要参与整

个传输过程，这样不会浪费发送端设备的资源。另外，即使接收端设备关机或出现故障，点到点传输也可以采用存储转发技术进行缓冲。点到点传输的缺点是发送端发出数据后，不知道接收端能否收到或何时能收到数据。IP 及以下各层采用的是点到点传输。

⑧ 端到端：体现在逻辑上的两两连接。端到端是体现在网络传输层之间的，比如要将数据从A传送到E，中间可能经过A→B→C→D→E，对于传输层来说，它并不知道B、C、D的存在，它只认为报文数据是从A直接到E的，这就叫做端到端。总之，端到端是由无数的点到点实现和组成的。

2. 网络分层模型

分层能使复杂的问题简单化，网络分层也是基于此原理。网络分层简化了网络设计，提高了网络互联的标准化程度。网络分层是上一层都依赖于下一层，只有最低层才是物理的实际通信，其他对等层是虚拟通信。分层原理与方法如图 17-1 所示，网络分层模型涉及以下术语。

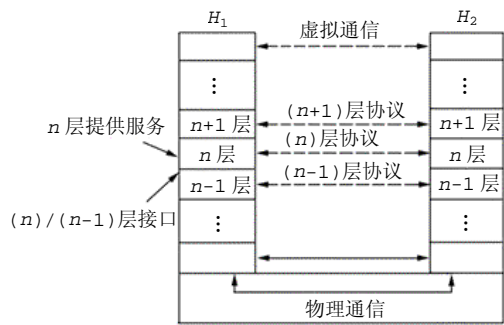


图 17-1 网络分层模型图

- ① 实体：每一层中的活动元素，可以是任何可发送或接收信息的硬件或软件进程。许多情况下，实体就是一个特定的软件模块。
- ② 对等实体：位于不同机器上同一层中的实体。
- ③ 服务提供者：*n* 层（下层）是 *n+1* 层（上层）的服务提供者。
- ④ 服务用户：*n+1* 层（上层）是 *n* 层（下层）的服务用户。
- ⑤ 服务访问点（SAP）：服务提供的地点，也即接口上相邻两层实体交换信息之处。
- ⑥ 服务和协议的关系：服务是垂直的，协议是水平的。*n* 层的服务用户只能看见 *n* 层的服务，无法看见 *n* 层的协议，在 *n* 层协议控制下，两个对等实体间的通信使得 *n* 层能够向 *n+1* 层提供服务，要实现 *n* 层协议，需要使用 *n-1* 层提供的服务。

互联网的两大网络参考模型（OSI 参考模型和 TCP/IP 参考模型）都是基于分层原理实现的。通过网络分层可以获得的好处有：各层之间相互独立，相邻层间交互只通过接口，使整个问题的复杂度下降。结构上可分隔开，各层都可以采用最合适的技术来实现，每一层的功能简单，易于实现和维护。若某一层改动时，只要不改变接口服务的关系，其他层则不受影响，灵活性好。分层有利于促进网络协议的标准化。

3. OSI 网络模型

(1) OSI 分层模型

开放式系统互联模型（OSI）是 1984 年由国际标准化组织（ISO）提出的一个网络参考模型。作为一个概念性框架，提出时希望以后不同的设备制造商和应用软件开发商遵循此标准。现在，此模型已成为计算机间和网络间进行通信的主要模型，目前使用的大多数网络通信协议的结构都是基于 OSI 参考模型或参照 OSI 参考模型的。

OSI模型将网络分为七层，即物理层、数据链路层、网络层、传输层、会话层、表示层和应用层，如图 17-2 所示。

应用层
表示层
会话层
传输层
网络层
数据链路层
物理层

图 17-2 OSI 参考模型

对 OSI 分层模型中各层的解释如下：

① 物理层（Physical layer）是参考模型的最低层。该层是网络通信的数据传输介质，由连接不同结点的电缆与设备共同构成。物理层规定了激活、维持、关闭通信端点之间的机械特性、电气特性、功能特性以及过程特性。该层为上层协议提供了一个传输数据的物理媒体。在这一层，数据的单位称为比特（bit）。

② 数据链路层（Data link layer）是参考模型的第 2 层。主要功能是：在物理层提供服务的基础上，在通信的实体间建立数据链路连接，传输以“帧”为单位的数据包，并采用差错控制与流量控制方法，使有差错的物理线路变成无差错的数据链路。

③ 网络层（Network layer）是参考模型的第 3 层。主要功能是：为数据在结点之间传输创建逻辑链路，通过路由选择算法为分组通过通信子网选择最适合的路径，以及实现拥塞控制、网络互联等功能。

④ 传输层（Transport layer）是参考模型的第 4 层。主要功能是：向用户提供可靠的端到端服务，处理数据包错误、数据包次序，以及其他一些关键的传输问题。传输层向高层屏蔽了下层数据通信的细节，因此，它是计算机通信体系结构中关键的一层。

⑤ 会话层（Session layer）是参考模型的第 5 层。主要功能是：负责维护两个结点之间的传输链接，以确保点到点的传输不中断，以及管理数据交换等功能。

⑥ 表示层（Presentation layer）是参考模型的第 6 层。主要功能是：用于处理在两个通信系统中交换信息的表示方式，主要包括数据格式变换、数据加密与解密、数据压缩与恢复等功能。

⑦ 应用层（Application layer）是参考模型的最高层，为操作系统或网络应用程序提供访问网络服务的接口。

(2) OSI 参考模型的特点

OSI 参考模型属于分层网络互联模型，分为通信子网和资源子网两级结构。

OSI 参考模型只有物理层之间是直接连接的，对等层之间采用相同的对等协议。该模型在发送数据时，数据从高层到低层；在接收数据时，数据从低层到高层。

4. 网络分层数据流说明

图 17-3 画出了网络分层时的数据流图。网络中的各层把数据当做一个流来处理，每层都有自己的传输单位，物理层的传输单位是比特流，而只有这一层是物理的数据传输，其他层都是逻辑的；链路层的传输单位是帧；网络层的传输单位是分组；传输层的传输单位是段。源主机应用层数据往下层传递时，每一层要增加相应的首部，称为封装。到达目的主机后，数据往上传递时需要再剥掉相应的首部，称为拆封。

17.1.2 TCP/IP 模型

1. TCP/IP 协议介绍

TCP/IP（又称为TCP/IP协议簇）是一组用于实现网络互联的通信协议，其名称来源于该协议簇中两个重要的协议（IP协议和TCP协议）。基于TCP/IP的参考模型将协议分成四个层次，它们分别是网络接口层、网际互联层（IP层）、传输层（TCP层）和应用层。图 17-4 画出了TCP/IP模型以及该模型与OSI模型各层的对照关系。

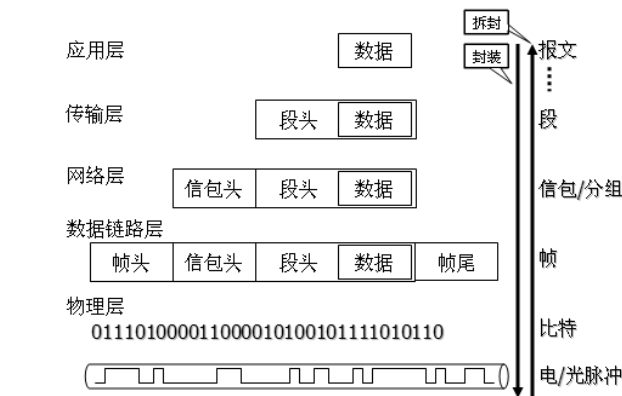


图 17-3 网络分层数据流图

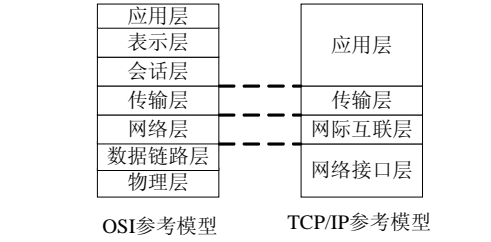


图 17-4 OSI 参考模型与 TCP/IP 参考模型对照图

- ① 网络接口层：网络接口层与 OSI 参考模型中的物理层和数据链路层相对应。
- ② 网际互联层：网际互联层对应于OSI参考模型的网络层，主要解决主机到主机的通信问题。该层有四个主要的协议：网际协议（IP）、地址解析协议（ARP）、互联网组管理协议（IGMP）和互联网控制报文协议（ICMP）。IP协议是网际互联层最重要的协议，它提供的是一个不可靠、无连接的数据报传递服务。
- ③ 传输层：传输层对应于 OSI 参考模型的传输层，为应用层实体提供端到端的通信功能。该层定义了两个主要的协议：传输控制协议（TCP）和用户数据报协议（UDP）。TCP 协议提供的是一种可靠的、面向连接的数据传输服务；而 UDP 协议提供的是不可靠的、无连接的数据传输服务。
- ④ 应用层：应用层对应于 OSI 参考模型的会话层、表示层和应用层，为用户提供所需要的

各种服务，例如 FTP、Telnet、DNS、SMTP 等。

2. OSI 参考模型与 TCP/IP 参考模型比较说明

(1) OSI 参考模型与 TCP/IP 模型比较说明

OSI 引入了服务、接口、协议、分层的概念，TCP/IP 借鉴了 OSI 的这些概念建立 TCP/IP 参考模型。OSI 先有模型，后有协议，先有标准，后进行实践；而 TCP/IP 则相反，先有协议和应用，再提出了模型，且是参照的 OSI 参考模型。OSI 太复杂，TCP/IP 简单却并不全面。OSI 花了很长时间进行标准化，与此同时，TCP/IP 已被广泛使用，并已成为网络互联事实上的标准。

(2) OSI 参考模型与 TCP/IP 参考模型

表 17-1 列出了 OSI 参考模型与 TCP/IP 参考模型的协议对照及协议说明。

表 17-1 OSI 参考模型与 TCP/IP 参考模型协议对照表

OSI 中的层	功 能	TCP/IP 协议簇
应用层	文件传输、电子邮件服务、文件服务、虚拟终端服务	TFTP、HTTP、SNMP、FTP、SMTP、DNS、Telnet
表示层	数据格式化、代码转换、数据加密	没有协议
会话层	解除或建立与别的接点的联系	没有协议
传输层	提供端到端的接口	TCP、UDP
网络层	为数据包选择路由	IP、ICMP、OSPF、BGP、IGMP 、ARP、RARP
数据链路层	传输有地址的帧以及错误检测功能	SLIP、PPP、MTU
物理层	以二进制数据形式在物理媒体上传输数据	ISO 2110、IEEE 802、IEEE 802.2

3. TCP/IP 通信用程图

图 17-5 画出了两台主机通过 TCP/IP 协议的通信用程，其中应用层使用的是 FTP 协议。除物理层是实际传输数据的，其他对等层只是虚拟通信（所以画的是虚线），对等层一般需要遵照相同的协议。上层传输数据只与相邻上下层有关系。TCP 及以下层是由操作系统内核实现的，即是处理通信细节的。而应用层程序是通过 Socket 编程实现的，处理的是应用程序细节。

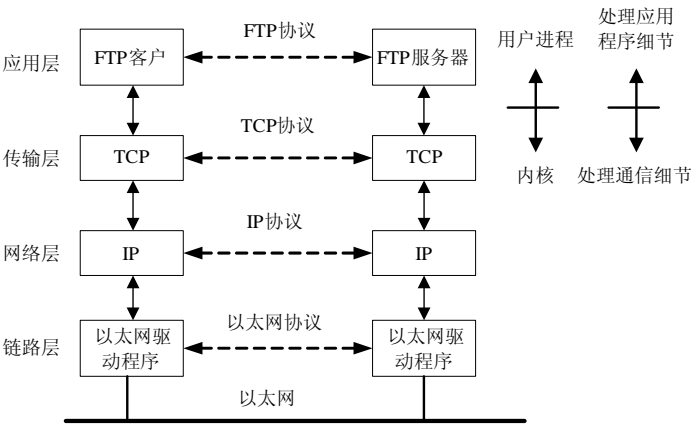


图 17-5 TCP/IP 通信用程图

4. TCP/IP 数据包

TCP/IP 数据包说明如图 17-6 所示。IP 层传输单位是 IP 分组，属于点到点的传输；TCP 层传输单位是 TCP 段，属于端到端的传输。

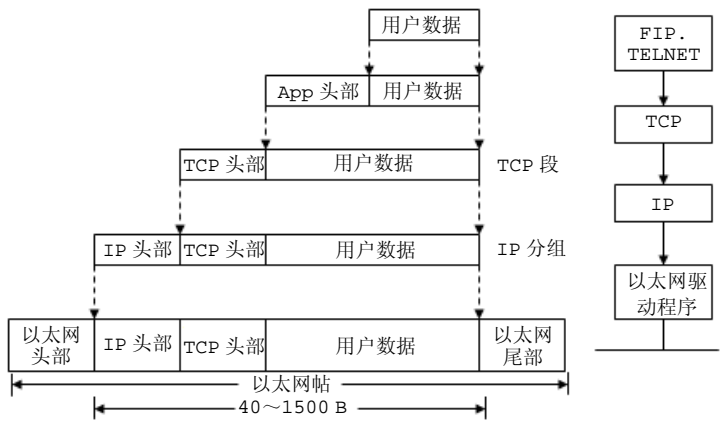


图 17-6 TCP/IP 数据包说明图

17.1.3 网络分类与广域网

1. 网络分类

通过网络分类，可以更好地了解网络的特征和特点。下面按网络八个方面的特征对网络进行分类，并给出了简要说明。

① 按覆盖范围分，可分为如下三种。

局域网（LAN）：如企业内部网、家庭内部网。

城域网（MAN）：一个城市的内部网络。

广域网（WAN）：连接世界各地方的网络。

② 按拓扑结构，可分为如下五种，如图 17-7 所示。

星形拓扑：交换机网络，如企业网络。

总线形拓扑：以太网，如企业部门级网络。

环形拓扑：令牌环网，如 FDDI 网。

树形拓扑：层次级网络，如层次级城域网。

网状形拓扑：广域网。

③ 按信息的交换方式，可分为如下三种。

电路交换：如打电话时，两端通话期间占用一条单独的线路。传统的电话线路属于电路交换。

报文交换：通信两端在一个时间段内传递一个报文。

报文分组交换：通信两端在一个时间段内传递一个报文分组，最后由通信双方合成报文。

TCP/IP 网络就属于报文分组交换。

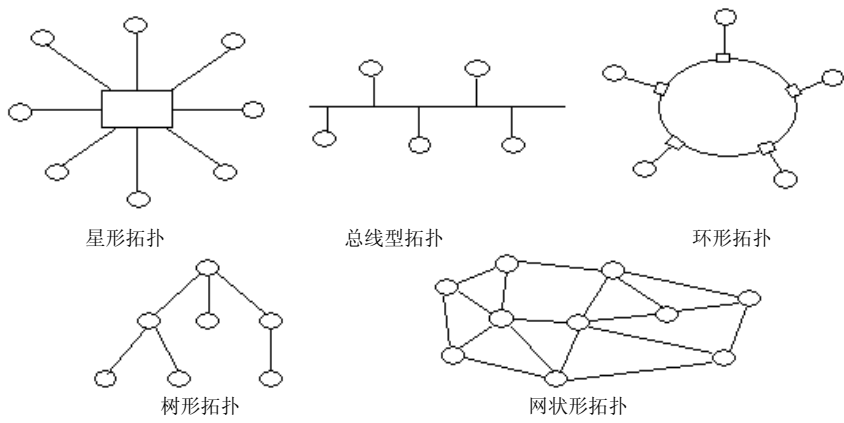


图 17-7 网络拓扑图

④ 按传输介质，可分为如下三类。

有线网：利用同轴电缆或双绞线介质通信的网络，如企业局域网。

光纤网：利用光纤进行通信的网络，如国家骨干网。

无线网：利用微波、磁波通信的网络，如手机 GSM 网、3G 网和 WiFi 网。

⑤ 按通信方式，可分为如下两类。

点对点传输网络：交换机网络。

广播式传输网络：以太网网络。

⑥ 按照网络用途，可分为通信子网和资源子网，如图 17-8 所示。

通信子网：计算机网络中负责数据通信的部分，它属于 OSI 模型的下三层（物理层、链路层、网络层）。其中，网络层的主要任务是通过路由算法，为分组通过子网选择最合适的路径。通信子网一般由通信介质（光纤、铜缆、双绞线）、交换机、路由器组成。通信子网具有以下三个特征：在通信子网中，用户数据被截成一定长度的分组来传输；通信需经中间路由器存储转发，每一个中间路由器从其输入线路上将分组完整地接收下来，存在缓冲区里，为它选择一条合适的输出线路，待该线路空闲时，把分组转发出去；通信子网一般叫做点-点子网、存储-转发子网或分组交换子网，大多数广域网的通信子网都是存储-转发子网。

资源子网：面向用户的部分，负责全网络面向应用的数据处理工作，而通信双方遵守共同协议。资源子网由计算机硬件和通信软件组成。

⑦ 按网络使用目的，可分为共享资源网、数据处理网和数据传输网。

⑧ 按服务方式，可分为客户端/服务器网络和对等网。

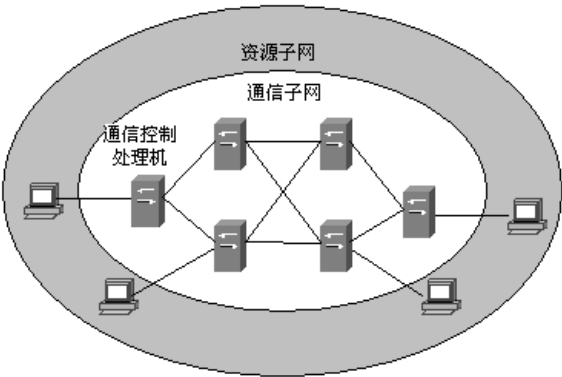


图 17-8 通信子网与资源子网划分图

2. 广域网

因特网（Internet）是一种连接全球的开放式广域网。

在因特网中，主机到主机的通信可能经过多种互联网服务提供商（Internet Service Provider，即 ISP），如图 17-9 所示。



图 17-9 广域网 ISP 图

广域网物理连接如图 17-10 所示，它是一种网状连接结构，一般是两两相连、一点与多点或多点与多点相连。



图 17-10 广域网物理连接图

互联网按照连接范围分，可分为局域网、接入网、城域网和广域网，如图 17-11 所示。局域网一般指家庭网络或企业网络等小范围的局部网络；接入网指接入电信服务商的网络；城域网指一个城市的网络；广域网指城市之间、国家之间的网络。

广域网从组网逻辑结构上可分为接入层、汇聚层和核心层，如图 17-12 所示。接入层由低端路由器提供对网络设备的接入，汇聚层由中端路由器或交换机对接入层数据包进行汇聚，核心层由骨干路由器负责数据包的路由选择和高速转发。

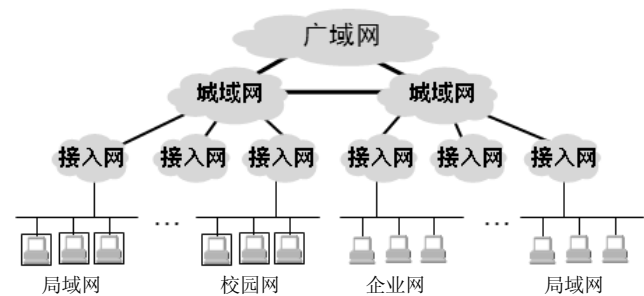


图 17-11 广域网、城域网以及局域网的关系

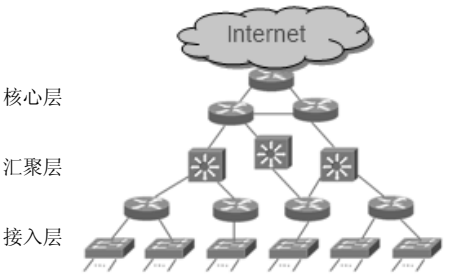


图 17-12 广域网组网逻辑结构图

17.1.4 网络地址

1. 网络地址分类

在网络中，按照用途可把网络地址分为如下四类。

- ① 物理地址：即 MAC 地址，如 00-aa-00-62-c9-09。
- ② 逻辑地址：即 IP 地址，如 127.0.0.1。
- ③ 端口地址：应用进程端口号。
- ④ 域名地址：万维网地址，如 www.google.com。

其中 MAC 地址为 48 位，0~23 位叫做组织唯一标识符，由 IEEE（电气和电子工程师协会）管理，生产以太网网卡的厂家就购买其中一组，再自行分配后 24 位，逐个将唯一地址赋予以太网卡。MAC 地址属于平面地址，无层次结构，每个网络设备的网络地址全球唯一，只能用于局部范围寻址，MAC 地址存在于网络的数据链路层。

2. IP 地址

IP 地址标识着网络中一个主机的位置，每个 IP 地址由 32 位（IPv4 地址，4 个字节）组成。分为网络号、主机号两部分，但现在都是采用 CIDR 方案，所以，IP 地址现在只表示主机号。另外，IP 地址也是全球唯一的，寻址很容易。

IP 地址有两种表示形式，分为二进制数表示和点分十进制数表示，其中二进制数由计算机内部使用，点分十进制数是方便人们记忆而引入的。

IP 地址按照用途分，可分为单播地址、广播地址和多播（组播）地址，其中广播地址和多播地址仅应用于 UDP 协议。

3. IP 地址分类

IPv4 的 IP 地址长度为 4B，通常采用点分十进制数表示法。例如，二进制数 IP 地址 0xc0a80002

用点分十进制数可表示为 192.168.0.2。Internet 被各种路由器和网关设备分隔成很多网段，为了标识不同的网段，需要把 32 位的 IP 地址划分成网络号和主机号两部分，网络号相同的各主机位于同一网段，相互间可以直接通信，网络号不同的主机之间通信则需要通过路由器转发。

最早的 IP 地址分类没有单独的网络号，将 IP 地址分为五类，如图 17-13 所示，以适应大型、中型、小型网络的需要。



图 17-13 IP 地址分类图

其中：

- A 类：范围是 0.0.0.0 到 127.255.255.255。
- B 类：范围是 128.0.0.0 到 191.255.255.255。
- C 类：范围是 192.0.0.0 到 223.255.255.255。
- D 类：范围是 224.0.0.0 到 239.255.255.255。
- E 类：范围是 240.0.0.0 到 247.255.255.255。

一个 A 类网络可容纳的地址数量最大，能容纳 1677 万多台电脑，一个 B 类网络的地址数量是 65536，一个 C 类网络的地址数量是 256，D 类地址用做多播地址，E 类地址保留未用。

随着 Internet 的飞速发展，这种划分方案的局限性很快显现出来，大多数组织都申请 B 类网络地址，导致 B 类地址很快就分配完了，而 A 类却浪费了大量地址。这种方式对网络的划分是扁平结构（flat）的，而不是层级结构（hierarchical）的，Internet 上的每个路由器都必须掌握所有的网络信息，随着 C 类网络的大量出现，路由器需要检索的路由表越来越庞大，负担越来越重。

由于用一个 IP 地址来表示网络号和主机号造成了上面所述的问题，IEEE 便提出了新的划分方案，即无类域间路由（Classless InterDomain Routing，CIDR）方案。网络号和主机号的划分需要用一个额外的子网掩码（subnet mask）来表示，而不能由 IP 地址本身的数值决定。也就是说，网络号和主机号的划分与这个 IP 地址是 A 类、B 类还是 C 类无关，因此，称为无类型的。

4. CIDR 子网划分方法

使用 CIDR 方案后，一个网络地址可包含 IP 地址和子网掩码两部分。例如，一个网络 IP 地

址为 140.252.20.68，子网掩码为 255.255.255.0，也可用简洁的表示方法，如 140.252.20.68/24。

进行自定义子网时，子网掩码需要由前面是连续的 1 和后面是连续的 0 构成。IP 地址与子网掩码做“与”运算可以得到网络号。

（1）子网划分用例 1

表 17-2 列出了子网划分用例 1，前面 24 位为网络号，后面 8 位是主机号，最多能连 254 台主机，因为 140.252.20.255 为该网段的广播地址。

表 17-2 子网划分用例 1

地址名称	IP 地址点分十进制数表示	IP 地址二进制数表示
IP 地址	140.252.20.68	8C FC 14 44
子网掩码	255.255.255.0	FF FF FF 00
网络号	140.252.20.0	8C FC 14 00
子网地址范围	140.252.20.0~140.252.20.255	

（2）子网划分用例 2

表 17-3 列出了子网划分用例 2，此用例前面 28 位为网络号，后面 4 位为主机号，最多能连 15 台主机。

表 17-3 子网划分用例 2

地址名称	IP 地址点分十进制数表示	IP 地址二进制数表示
IP 地址	140.252.20.68	8C FC 14 44
子网掩码	255.255.255.240	FF FF FF F0
网络号	140.252.20.64	8C FC 14 40
子网地址范围	140.252.20.64~140.252.20.79	

17.2 TCP/IP 协议簇报文格式

本节列出了常见的 TCP/IP 协议簇报文格式，如果使用原始套接字进行编程，将会用到下面的一些格式。

1. 以太网（RFC 894）帧格式

以太网帧是被网卡识别的帧格式，其格式遵循 RFC 894 协议，如图 17-14 所示。

其中的源地址和目的地址是指网卡的硬件地址（也叫 MAC 地址），长度是 48 位，是在网卡出厂时固化的。协议字段有三种值，分别对应 IP、ARP、RARP。以太网帧中的数据长度规定最小为 46B，最大为 1500B，若不足 46B，则要在后面补填充位，最大值 1500B 称为以太网的最大传输单元（MTU）。

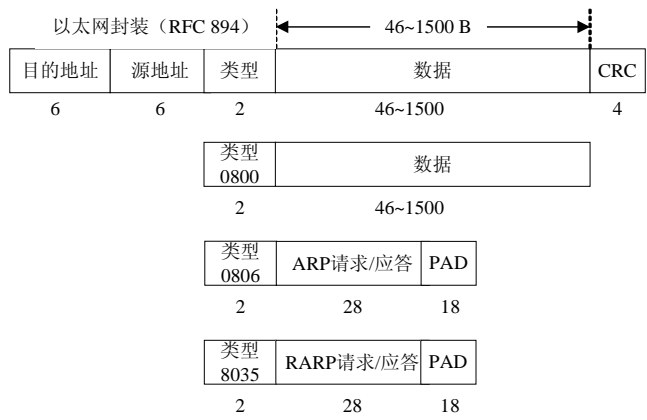


图 17-14 以太网帧格式图

2. IP 数据报文格式

IP 数据报文处于网络层，其报文中的目的 IP 地址供路由器进行路由选择。IP 报文格式如图 17-15 所示。

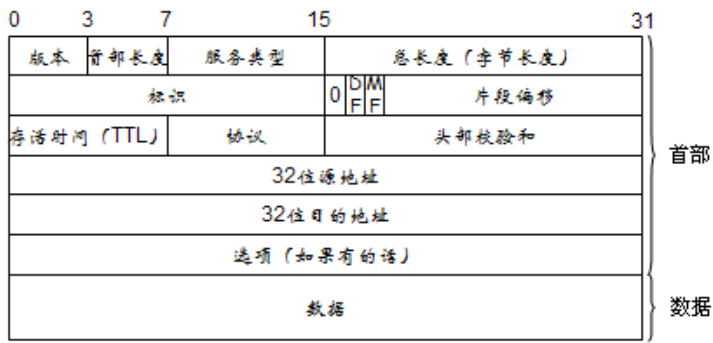


图 17-15 IP 数据报文格式图

对 IP 数据报文格式简要说明如下：IP 数据报的首部长度是可变长的，但总是 4B 的整数倍。对于 IPv4，4 位版本字段是 4。4 位首部长度的数值是以 4B 为单位的，最小值为 5，也就是说，首部长度最小是 4B×5=20B，也就是不带任何选项的 IP 首部，4 位能表示的最大值是 15，即首部长度最长是 60B。8 位服务类型字段有 3 位用来指定 IP 数据报文的优先级，4 位表示可选的服务类型（最小延迟、最大吞吐量、最大可靠性、最小成本），还有一位总是 0。总长度字段是整个数据报的字节数，16 位的标识用于分片和重组，协议字段说明上层协议的类型（如 TCP、UDP、ICMP 等）。

3. ICMP 报文格式

每个 ICMP 报文格式不尽相同，但前三个字段是相同的，即 1B 的 ICMP 类型 (TYPE)、1B 的代码 (CODE)、2B 的校验和 (CHECKSUM)，另外，报告差错的 ICMP 报文还包含产生问题的数据报首部及开头 64 比特数据。其报文如图 17-16 所示。

TYPE	CODE
CHECKSUM	
其他数据	

图 17-16 ICMP 数据报文格式图

表 17-4 列出了 ICMP 报文格式常见类型及其说明。

表 17-4 ICMP 报文格式类型及说明

种 类	ICMP 报文类型说明
0	回送应答
3	目的地不可达
4	源站抑制
5	重定向
8	回送请求
11	数据报超时
12	数据报参数错
13	时间戳请求
14	时间戳应答
17	地址掩码请求
18	地址掩码应答

4. UDP 报文格式

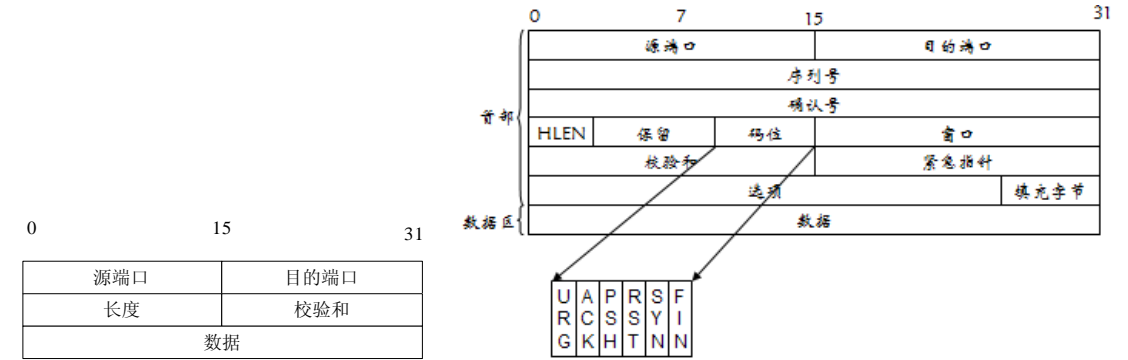
UDP 用户数据报协议提供无连接服务。UDP 缺乏可靠性支持，应用程序需要自行保证数据的可靠性，如用 UDP 协议实现可靠传输，必须自行实现确认、超时、重传、流控等功能。

UDP 数据报文格式很简单，因为它是无连接的面向记录的服务，一次只传一个报文，而且并不保证对方是否能收到，因此，也是非可靠的。其数据段格式如图 17-17 所示。

5. TCP 报文格式

TCP 传输控制协议是面向连接可靠的传输协议。TCP 通过给所发送数据的每一个段管理一个序列号进行排序，实现了丢失重传功能。TCP 通过通告窗口、拥塞窗口实现流量控制、RTT 估算、拥塞控制功能，同时 TCP 的连接是全双工的。TCP 递给 IP 的数据块叫做消息段（segment），这个消息段由 TCP 协议标题域（TCP header field）和存放应用程序的数据域（header fields）组成。

TCP 协议为可靠的连接协议，所以其格式复杂，它是网络端到端数据可靠传输的基础。其数据段格式如图 17-18 所示。



TCP 数据段格式说明如下：

- ① 源端口：说明发送端应用程序的端口。
- ② 目的端口：说明接收端应用程序的端口。
- ③ 序列号：封包序号，说明该包在封装数据流中的位置。

④ 确认号：回应序号，当接收端接收到 TCP 封包并通过检验确认之后，会依照发送序号，再加上数据长度产生一个回应序号，附在下一个响应封包送回给对方，这样接收端就知道刚才的封包已经被接收端成功接收到。

⑤ HLEN：首部长度，TCP 数据段首部包括固定和变长两部分。

⑥ 窗口：为通告窗口。

⑦ URG 位：如果使用紧急数据指针，则将这一位设为 1。

⑧ ACK 位：如果确认序列号有效，则设为 1。

⑨ PSH 位：表示“推”数据，如果这一位设置成 1，表示希望接收方在接收到这个数据段之后，将它立即传送给高层应用程序，而不是缓存起来。

⑩ RST 位：表示请求重置连接。当 TCP 协议接收到一个不能处理的数据段时，向对方 TCP 协议发送这种数据段，表示这个数据段所标识的连接出现了某种错误，请求对方 TCP 协议将这个连接清除。有三种情况可能导致 TCP 协议发送 RST 数据位：SYN 数据段指定的目的端口处没有接收进程等待；TCP 协议想放弃一个已经存在的连接；TCP 接收到一个数据段，但是这个数据段所标识的连接不存在。接收到 RST 数据位的 TCP 协议立即将这条连接非正常断开，并向应用程序报告。

⑪ SYN 位：请求建立连接。TCP 用这种数据段向对方 TCP 协议请求建立连接，在这个数据段中，TCP 协议将它选择的初始序列号通知对方，并且与对方协议协商最大数据段的大小。

⑫ FIN 位：请求关闭连接。当协议收到对这个数据段的确认后，关闭写方向的连接，因为 TCP 连接是全双工的，在发送了 FIN 数据段之后，它仍能接收数据，直至对方也发送 FIN 数据段。

第 18 章

socket编程

socket 通常也称为“套接字”，应用程序通常通过“套接字”向网络发出请求或者响应网络请求。socket 位于传输层之上、应用层之下。socket 函数基本为系统调用，它是操作系统向网络通信应用程序提供的函数接口。

18.1 套接字说明及函数说明

Linux 套接字编程完全兼容 UNIX 套接字编程，两个平台的编程方法是一致的。读者可以重点掌握 TCP 套接字编程和 UDP 服务器编程。

18.1.1 套接字说明

Linux 和 UNIX 的 I/O 内涵是系统中的一切都是文件。当程序在执行任何形式的 I/O 时，程序是在读或者写一个文件描述符，从而实现操作文件，但是，这个文件可能是一个 socket 网络连接、目录、FIFO、管道、终端、外设、磁盘上的文件。socket 是使用标准 Linux 文件描述符和其他进程进行通信的。

利用 socket 编程，会涉及流（stream）、连接（connection）、阻塞（block）、非阻塞（non-block）、同步（synchronous）、异步（asynchronous）、长连接、短连接、IP 地址、端口、字节顺序等概念，这些概念将在后文加以介绍。

1. 套接字说明

（1）套接字网络层次说明

图 18-1 画出了套接字位于网络中的层次，它位于传输层之上、应用层之下。socket 编程是通过一系列系统调用完成应用层协议。如 FTP、Telnet、HTTP 等应用层协议都是通过 socket 编程来实现的。

套接字是对网络中应用进程之间进行双向通信的抽象，它提供了应用层进程利用网络协议栈交换数据的机制。

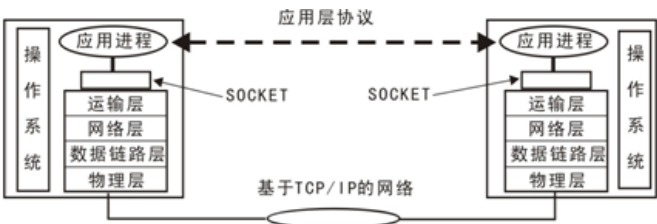


图 18-1 套接字层次图

从套接字所处的位置来讲，套接字上连应用进程，下连网络协议栈，是应用程序通过网络协议栈进行通信的接口，同时还是应用程序与网络协议栈进行交互的接口。

从实现的角度来讲，套接字系统函数是一个复杂的软件模块，包含了一定的数据结构和许多选项，由操作系统内核管理。

从使用的角度来讲，利用套接字函数编程非常简单。

总之，套接字是网络通信的基石。套接字函数是一种网络通信 API，程序员可以用它开发网络程序。

(2) IP 地址与端口

端口是指网络中面向连接服务和无连接服务的通信协议端口，它是一种抽象的软件结构，包括一些数据结构和 I/O（基本输入/输出）缓冲区。

IP 地址用来标识网络中不同主机的地址，而端口号就是标识同一台主机上不同网络通信进程的地址，IP 地址和端口号合起来标识网络中唯一的进程。

如果把IP地址比做一间房子，端口就是出入这间房子的门。真正的房子只有几个门，但是一个IP地址的端口可以有 65536（即 2^{16} ）个之多。端口是通过端口号来标记的，端口号只能是整数，范围为 $0 \sim 65535$ ($2^{16}-1$)。其中端口 1~1024 是系统保留端口。

一次 socket 通信连接会涉及源 IP 地址、源端口、目的 IP 地址、目的端口四个要素。源 IP 地址、源端口标识客户端进程，其中源端口是操作系统随机分配的；目的 IP 地址、目的端口标识服务端进程，其中目的端口是由应用程序指定的。

(3) socket 编程说明

Linux 系统是通过提供套接字（socket）函数来进行网络编程的。socket 技术提供了在 TCP/IP 模型各个层上的编程支持，该技术是在内核处理收到的各层协议数据，然后应用程序以文件操作方式接收内核传来的数据。应用程序对文件处理是通过一个文件描述符来进行的，socket 文件描述符可以看成普通的文件描述符来进行操作，这就是 Linux 设备无关性的好处，即可以通过对文件描述符的读写操作来实现网络间数据流的传输。

2. TCP 连接与关闭

(1) 建立 TCP 连接

建立 TCP 连接时，服务器和客户端需要通过三路握手才能正式建立连接。图 18-2 画出了客

户端与服务端三路握手的实现原理。

TCP 连接的过程说明如下：

- ① 服务器绑定端口进行通信侦听，通过调用 `socket`、`bind`、`listen` 函数完成。
- ② 客户端通过调用 `connect` 进行发起对服务端的连接，客户端发送一个 `SYN` 分节 `x` 告诉服务器客户端使用的端口以及 `TCP` 连接的初始序号。
- ③ 服务器在收到客户端的 `SYN` 报文后，将返回一个 `SYN+ACK` 的报文，`ACK` 分节 `x+1`（请求序号加 1）表示客户端的请求被接受，`SYN` 分节 `y` 为服务器请求客户端进行确认。
- ④ 客户端发送 `ACK` 分节 `y+1` 对服务端 `SYN` 分节 `y` 进行确认，此时三路握手完成。

(2) TCP 连接关闭

TCP 连接关闭时，双方需要通过四次会话，其流程如图 18-3 所示。

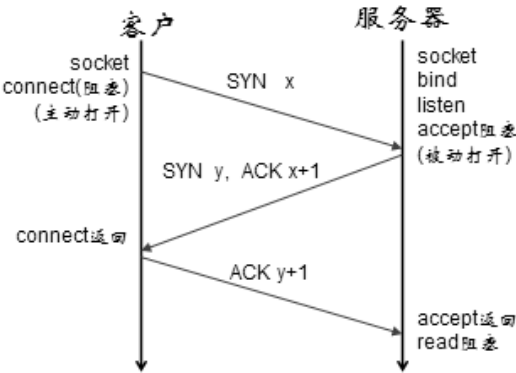


图 18-2 TCP 三路握手图

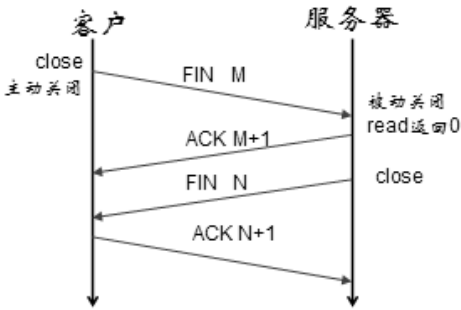


图 18-3 TCP 连接关闭图

TCP 一般需要用四个分节终止一个连接，具体流程描述如下：

- ① TCP 客户端发送一个 `FIN` 分节 `M`，用来关闭客户到服务器的数据传送。
- ② 服务器收到这个 `FIN`，它发回一个 `ACK`，确认序号为收到的 `M` 序号加 1。
- ③ 服务器开始关闭客户端的连接，发送一个 `FIN` 分节 `N` 给客户端。
- ④ 客户端发回 `ACK` 报文确认，并将确认序号设置为收到序号 `N` 加 1。

(3) TCP 连接中的分组交换

图 18-4 是 TCP 连接中的分组交换图，它画出了客户端和服务端连接的全过程，并标明通信过程中的连接状态。其中，`TIME_WAIT` 状态是执行主动关闭的那一端（即图 18-4 的客户端）进入这种状态，在该状态的持续时间是 `2MSL`（最长分节生命周期），这个状态的存在是为了保证 `ACK` 分节发送的成功。

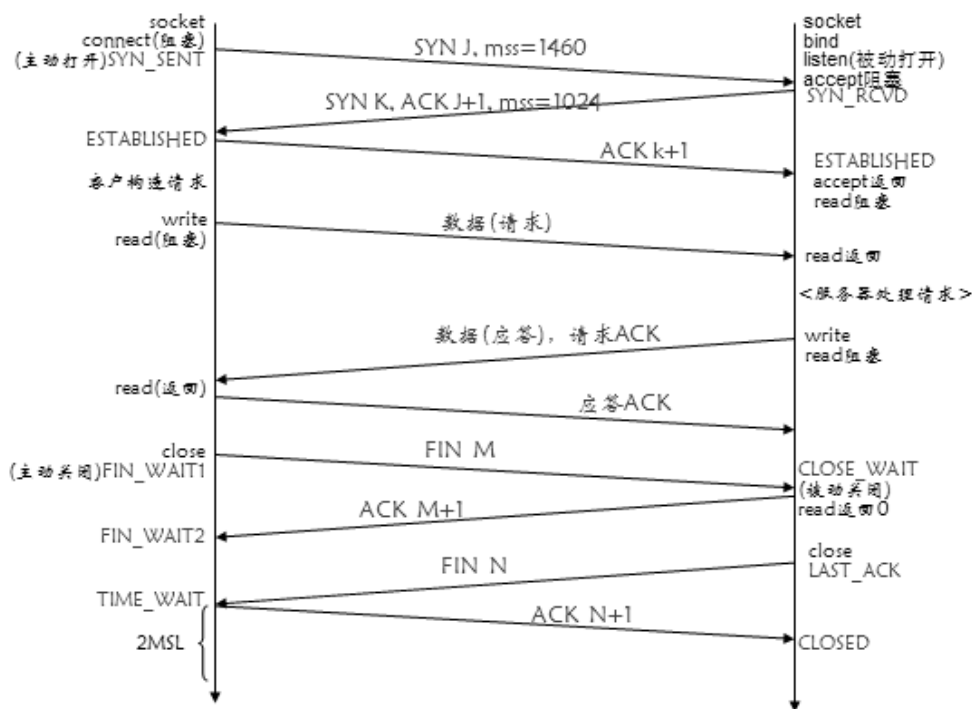


图 18-4 TCP 连接中的分组交换

18.1.2 socket 地址说明及转换函数

1. 三种常见的结构类型

在套接字编程中，有三种常见的结构类型，它们用来存放 socket 地址信息。这三种结构类型分别为 `struct in_addr`、`struct sockaddr`、`struct sockaddr_in`，对这三种结构类型说明如下。

`struct in_addr` 专门用来存储 IP 地址，对于 IPv4 来说，IP 地址为 32 位无符号整数，其定义如下：

```
struct in_addr {
    unsigned long s_addr;
};
```

`struct sockaddr` 结构用来保存套接字地址信息，其定义如下：

```
struct sockaddr {
    unsigned short sa_family; /* 地址簇, AF_xxx */
    char sa_data[14]; /* 14 B 的协议地址 */
};
```

`struct sockaddr` 结构中 `sa_family` 成员说明的是地址簇类型，一般为“`AF_INET`”；而 `sa_data` 则包含远程主机的 IP 地址和端口等信息。

`struct sockaddr` 结构类型使用在 socket 相关的系统调用函数中，但这个结构 `sa_data`

字段可以包含较多的信息，不便于编程和对其进行赋值，因此，建立了 struct sockaddr_in 结构，该结构与 struct sockaddr 结构的大小相等，能更好地处理 struct sockaddr 结构中的数据。对 struct sockaddr_in 结构变量进行赋值完成后，在进行 socket 相关的系统调用时，再将 struct sockaddr_in 结构变量强制转换为 struct sockaddr 结构类型。

struct sockaddr_in 结构定义如下：

```
struct sockaddr_in { /* “in” 代表 “Internet” */
short int sin_family; /* Internet 地址族*/
unsigned short int sin_port; /* 端口号*/
struct in_addr sin_addr; /* Internet 地址*/
unsigned char sin_zero[8]; /* 填充 0（为了保持和 struct sockaddr 一样大小）*/
};
```

在实际应用的编程中，对套接字地址结构的使用方法和流程如下：

① 定义一个 struct sockaddr_in 结构变量，并将它初始化为 0，代码如下：

```
struct sockaddr_in myad;
memset(&myad,0,sizeof(struct sockaddr_in));
```

② 给这个结构变量赋值，代码如下：

```
myad.sin_family=AF_INET;
myad.sin_port=htons(8080);
myad.sin_addr.s_addr=htonl(INADDR_ANY);
```

③ 在进行函数调用时，将这个结构强制转换为 struct sockaddr 类型，代码如下：

```
bind(serverFd, (struct sockaddr *)& myad, sizeof(myad))
```

2. 整型字节序转换函数

为保证“大端”和“小端”字节序的机器之间能相互通信，需在发送多字节的整数时，将主机字节序转换成网络字节序，或将网络字节序转换为主机字节序。字节转换主要是针对整型进行的，字符型由于是单字节，所以不存在这个问题。整型字节序转换函数原型及其说明如下。

所需头文件	#include <netinet/in.h>
函数说明	完成网络字节序与主机字节序的转换
函数原型	uint16_t htons(uint16_t hostshort) /*短整型主机转换为网络字节序*/ uint32_t htonl(uint32_t hostlong) /*长整型主机转换为网络字节序*/ uint16_t ntohs(uint16_t netshort) /*短整型网络转换为主机字节序*/ uint32_t ntohl(uint32_t netlong) /*长整型网络转换为主机字节序*/
函数传入值	hostshort、hostlong：为转换前的主机字节序数值 netshort、netlong：为转换前的网络字节序数值
函数返回值	① htons、htonl 返回转换后的网络字节序数值 ② ntohs、ntohl 返回转换后的主机字节序数值
附加说明	h 表示主机，n 表示网络，s 表示短整数，l 表示长整数，to 表示转换

3. IP 地址转换函数

IP 地址转换函数是指完成点分十进制数 IP 地址与二进制数 IP 地址之间的相互转换。IP 地址转换主要由 `inet_pton`、`inet_addr` 和 `inet_ntoa` 这三个函数完成，这三个地址转换函数都只能处理 IPv4 地址，而不能处理 IPv6 地址。这三个函数的函数原型及其具体说明如下。

inet_addr（将点分十进制数 IP 地址转换为二进制数地址）	
所需头文件	<code>#include <sys/socket.h></code> <code>#include <netinet/in.h></code> <code>#include <arpa/inet.h></code>
函数说明	将点分十进制数 IP 地址转换为二进制数地址
函数原型	<code>in_addr_t inet_addr(const char *cp)</code>
函数传入值	<code>cp</code> : 点分十进制数 IP 地址，如“10.10.10.1”
函数返回值	成功：返回二进制数形式的 IP 地址
	失败：返回一个常值 <code>INADDR_NONE</code> （32 位均为 1）

inet_aton（将点分十进制数 IP 地址转换为二进制数地址）	
所需头文件	<code>#include <sys/socket.h></code> <code>#include <netinet/in.h></code> <code>#include <arpa/inet.h></code>
函数说明	将点分十进制数 IP 地址转换为二进制数地址
函数原型	<code>int inet_aton(const char *cp, struct in_addr *inp)</code>
函数传入值	<code>cp</code> : 点分十进制数 IP 地址，如“10.10.10.1”
函数传出值	<code>inp</code> : 转换后二进制数地址信息保存在 <code>inp</code> 中
函数返回值	成功：非 0
	失败：0

inet_ntoa（将二进制数地址转换为点分十进制数 IP 地址）	
所需头文件	<code>#include <sys/socket.h></code> <code>#include <netinet/in.h></code> <code>#include <arpa/inet.h></code>
函数说明	将二进制数地址转换为点分十进制数 IP 地址
函数原型	<code>char *inet_ntoa(struct in_addr in)</code>
函数传入值	<code>in</code> : 二进制数 IP 地址
函数返回值	成功：返回字符串指针，此指针指向了转换后的点分十进制数 IP 地址
	失败：NULL

4. 获取主机信息函数

（1）`getpeername` 函数原型

getpeername（获取通信对方信息）	
所需头文件	<code>#include <netinet/in.h></code>
函数说明	获取通信对方信息

续表

getpeername（获取通信对方信息）	
函数原型	int getpeername(int s, struct sockaddr *name, socklen_t *namelen)
函数传入值	s: 所连接的 socket 文件描述符
	namelen: 返回 name 的长度，可以设置为 sizeof(struct sockaddr)
函数传出值	name: 指向包含对方 IP 地址、端口等信息的 sockaddr 结构地址
函数返回值	成功: 0
	失败: -1, 失败原因存于 error 中

（2）gethostname 函数原型

gethostname（获取本地主机的名称）	
所需头文件	#include <netinet/in.h>
函数说明	返回本地主机的名称
函数原型	int gethostname(char *name, size_t len)
函数传入值	len: name 数组的长度
函数传出值	name: 主机的名称
函数返回值	成功: 0
	失败: -1, 失败原因存于 error 中

（3）gethostname 函数举例

gethostname.c 源代码如下：

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
int main()
{
    char hostname[30] ;
    int flag =0 ;
    memset(hostname, 0x00, sizeof(hostname));
    flag = gethostname(hostname, sizeof(hostname));
    if( flag <0 )
    {
        perror("gethostname error") ;
        return -1 ;
    }
    printf( "hostname = %s\n", hostname) ;
    return 0 ;
}
```

编译 gcc gethostname.c -o gethostname。

执行 ./gethostname，执行结果如下：

```
hostname = ubuntu
```

（4）通过主机名或域名获取 IP 地址

gethostbyname (通过主机名或域名获取网络信息)	
所需头文件	#include <netinet/in.h>
函数说明	<p>gethostbyname 会返回一个 hostent 结构指针，该结构具体说明如下：</p> <pre>struct hostent { char *h_name; /*正式的主机名称*/ char **h_aliases; /*该主机的其他别名*/ int h_addrtype; /*地址类型，通常是 AF_INET*/ int h_length; /*地址的长度*/ char **h_addr_list; /*该主机的所有地址*/ };</pre>
函数原型	struct hostent *gethostbyname(const char *name)
函数传入值	name: 域名或主机名
函数传出值	name: 主机的名称
函数返回值	成功: 返回 hostent 指针
	失败: NULL, 失败原因存于 h_error 中（注意：错误原因不存于 error 中）
附加说明	该函数首先在/etc/hosts 文件中查找是否有匹配的主机名，有则用匹配项解释此主机名；如果没有，则根据域名解析此主机名

(5) gethostbyname 函数举例

gethostbyname.c 源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
int main(int argc, char *argv[])
{
    struct hostent *h;
    if (argc != 2) { /* error check the command line */
        fprintf(stderr,"usage: getip address\n");
        return -1;
    }
    if ((h=gethostbyname(argv[1])) == NULL) { /* get the host info */
        perror("gethostbyname");
        return -1;
    }
    printf("Host name  : %s\n", h->h_name);
    printf("IP Address : %s\n",inet_ntoa(*((struct in_addr *)h->h_addr)));
    return 0;
}
```

编译 gcc gethostbyname.c -o gethostbyname。

执行./gethostbyname www.sina.com, 执行结果如下：

```
Host name  : libra.sina.com.cn
IP Address : 202.108.33.83
```

18.1.3 socket 主要函数说明

1. 基本套接字函数

(1) socket 函数原型

socket (建立一个 socket 文件描述符)		
所需头文件	#include <sys/types.h> #include <sys/socket.h>	
函数说明	建立一个 socket 文件描述符	
函数原型	int socket(int domain, int type, int protocol)	
函数传入值	domain	AF_INET: IPv4 协议
		AF_INET6: IPv6 协议
		AF_LOCAL: UNIX 域协议
		AF_ROUTE: 路由套接口
		AF_KEY: 密钥套接口
	type	SOCKET_STREAM: 双向可靠数据流, 对应 TCP
		SOCKET_DGRAM: 双向不可靠数据报, 对应 UDP
		SOCKET_RAW: 提供传输层以下的协议, 可以访问内部网络接口, 例如, 接收和发送 ICMP 报文
	protocol	type 为 SOCKET_RAW 时, 需要设置此值说明协议类型, 其他类型设置为 0 即可
函数返回值	成功: socket 文件描述符	
	失败: -1, 失败原因存于 error 中	

表 18-1 列出了当进行 socket 调用时, 其中的协议簇 (domain) 与类型 (type) 可能产生的组合。

表 18-1 socket 中协议簇 (domain) 与类型 (type) 组合表

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP	TCP	Yes		
SOCK_DGRAM	UDP	UDP	Yes		
SOCK_RAW	IPv4	IPv6		Yes	Yes

(2) bind 函数原型

bind (将一个本地协议地址与 socket 文件描述符联系起来)		
所需头文件	#include <sys/types.h> #include <sys/socket.h>	
函数说明	将一个协议地址与 socket 文件描述符联系起来	
函数原型	int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)	
函数传入值	sockfd	socket 文件描述符
	addr	my_addr 指向 sockaddr 结构, 该结构包含 IP 地址和端口等信息
	addrlen	sockaddr 结构的大小, 可设置为 sizeof(struct sockaddr)
函数返回值	成功: 0	
	失败: -1, 失败原因存于 error 中	

利用 bind 函数绑定地址时，可以指定 IP 地址和端口号，也可以指定其中之一，甚至一个也不指定。可以使用通配地址 INADDR_ANY（为宏定义，其值等于 0），它通知内核选择 IP 地址。表 18-2 列出了设置 socket 地址结构的几种方式，但在实际中，绑定的端口号都需要指定。

表 18-2 设置 socket 地址结构的几种方式

进程指定		说 明
IP 地址	端 口	
通配地址 INADDR_ANY	0	内核自动选择 IP 地址和端口号
通配地址 INADDR_ANY	非 0	内核自动选择 IP 地址，进程指定端口号
本地 IP 地址	0	进程指定 IP 地址，内核自动选择端口号
本地 IP 地址	非 0	进程指定 IP 地址和端口号

(3) listen 函数原型

listen（等待连接）		
所需头文件	#include <sys/types.h> #include <sys/socket.h>	
函数说明	等待连接	
函数原型	int listen(int sockfd, int backlog)	
函数传入值	sockfd	监听 socket 文件描述符
	backlog	套接字排队的最大连接个数
函数返回值	成功：0	
	失败：-1，失败原因存于 error 中	
特别说明	对于监听 socket 文件描述符 sockfd，内核要维护两个队列，分别为未完成连接队列和已完成连接队列，这两个队列之和不超过 backlog	

(4) connect 函数原型

connect（建立 socket 连接）		
所需头文件	#include <sys/types.h> #include <sys/socket.h>	
函数说明	建立 socket 连接	
函数原型	int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen)	
函数传入值	sockfd	socket 文件描述符
	serv_addr	连接的网络地址和端口
	addrlen	sockaddr 结构的大小，可设置为 sizeof(struct sockaddr)
函数返回值	成功：0	
	失败：-1，失败原因存于 error 中	
附加说明	函数 connect 激发 TCP 的三路握手过程，出错返回有以下几种情况： ① 如果客户没有收到 SYN 分节的响应（总共 75s，这之间可能重发了若干次 SYN），则返回 ETIMEDOUT ② 如果对客户的 SYN 的响应是 RST，则表明该服务器主机在指定的端口上没有进程在等待与之相连，函数返回错误 ECONNREFUSED ③ 如果客户发出的 SYN 在中间路由器上引发一个目的地不可达的 ICMP 错误，内核返回 EHOSTUNREACH 或 ENETUNREACH 错误（即 ICMP 错误）给进程	

(5) accept 函数原型

accept（接受 socket 连接）	
所需头文件	<code>#include <sys/types.h></code> <code>#include <sys/socket.h></code>
函数说明	接受 socket 连接，返回一个新的 socket 文件描述符，原 socket 文件描述符仍为 listen 函数所用，而新的 socket 文件描述符用来处理连接的读写操作
函数原型	<code>int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)</code>
函数传入参数	Socketfd: socket 文件描述符
	addrlen: addr 的大小，可设置为 sizeof(struct sockaddr)
函数传出参数	addr: 写入远程主机的地址数据
函数返回值	成功: 实际读取的字节数
	失败: -1, 错误代码存放在 error 中
附加说明	① accept 函数由 TCP 服务器调用，为阻塞函数，从已完成连接的队列中返回一个连接；如果该对列为空，则进程进入阻塞等待 ② 函数返回的套接字为已连接套接字，而监听套接字仍为 listen 函数所用

(6) close 函数原型

close（关闭连接的 socket 文件描述符）	
所需头文件	<code>#include <unistd.h></code>
函数说明	关闭连接的 socket 文件描述符
函数原型	<code>int close(int sockfd)</code>
函数传入值	sockfd: socket 文件描述符
函数返回值	成功: 0
	失败: -1, 失败原因存于 error 中
附加说明	① close 函数默认功能是将套接字置为“已关闭”标记，并立即返回给进程，这个套接字不能再为该进程所用 ② 正常情况下，close 将引发四个分节终止序列，但在终止前将发送已排队的数据 ③ 如果套接字描述符访问计数在调用 close 后大于 0（多个进程共享同一个套接字的情况下），则不会引发 TCP 终止序列（即不会发送 FIN 分节）

(7) shutdown 函数原型

shutdown（终止 socket 通信）		
所需头文件	<code>#include <sys/socket.h></code>	
函数说明	终止 socket 通信	
函数原型	<code>int shutdown(int s, int how)</code>	
函数传入值	s	socket 文件描述符
	how	0 (SHUT_RD): 关闭 socket 连接的读部分，不再接收套接字中的数据且现留在收缓冲区的数据作废
		1 (SHUT_WR): 关闭 socket 连接的写部分（半关闭），但留在套接字发送缓冲区中的数据都会被发送，后跟 TCP 连接终止序列，不管访问计数是否大于 0，此后将不能再执行对套接字的任何写操作
		2 (SHUT_RDWR): socket 连接的读、写都关闭
函数返回值	成功: 0	
	失败: -1, 失败原因存于 error 中	

(8) read 函数原型

read（从打开的 socket 文件流中读取数据）	
所需头文件	#include <unistd.h>
函数说明	从打开的 socket 文件流中读取数据，这里仅说明此函数应用于 socket 的情况
函数原型	ssize_t read(int fd, void *buf, size_t count)
函数传入参数	Fd: socket 文件描述符
	count: 读取的最大字节数
函数传出参数	buf: 读取数据的首地址
函数返回值	成功: 实际读取的字节数
	失败: -1, 错误代码存放在 error 中
附加说明	调用函数 read 从 socket 文件流中读取数据时，有如下几种情况： ① 套接字接收缓冲区接收数据，返回接收到的字节数 ② TCP 协议收到 FIN 数据，返回 0 ③ TCP 协议收到 RST 数据，返回-1，同时 errno 设置为 ECONNRESET ④ 进程阻塞过程中接收到信号，返回-1，同时 errno 设置为 EINTR

(9) write 函数原型

write（向 socket 文件流中写入数据）	
所需头文件	#include <unistd.h>
函数说明	向 socket 文件流中写入数据，这里仅说明此函数应用于 socket 的情况
函数原型	ssize_t write (int fd, const void *buf, size_t count)
函数传入参数	Fd: socket 文件描述符
	buf: 写入数据的首地址
	count: 写入的最大字节数
函数返回值	成功: 实际写入的字节数
	失败: -1, 错误代码存放在 error 中
附加说明	调用函数 write 向 socket 文件流写数据时，有如下几种情况： ① 套接字发送缓冲区有足够的空间，返回发送的字节数 ② TCP 协议接收到 RST 数据，返回-1，同时 errno 设置为 ECONNRESET ③ 进程阻塞过程中接收到信号，返回-1，同时 errno 设置为 EINTR

2. 高级套接字函数

recv 和 send 函数提供了与 read 和 write 差不多的功能。不过它们提供了一个控制参数 flags 来进行读写的额外控制操作。

(1) send 函数原型

send（通过 socket 文件描述符发送数据到对方）	
所需头文件	#include <sys/types.h>
	#include <sys/socket.h>
函数说明	通过 socket 文件描述符发送数据到对方
函数原型	ssize_t send(int s, const void *buf, size_t len, int flags)

续表

send（通过 socket 文件描述符发送数据到对方）		
函数传入值	s	socket 文件描述符
	buf	发送数据的首地址
	len	发送数据的长度
	flags	0：此时的功能同 write，flags 还可以设为以下标志的组合 MSG_OOB：发送带外数据 MSG_DONTROUTE：告诉 IP 协议，目的主机在本地网络，没有必要查找路由表 MSG_DONTWAIT：设置为非阻塞操作 MSG_NOSIGNAL：表示发送动作不愿被 SIGPIPE 信号中断
函数返回值	成功：实际发送的字节数	
	失败：-1，失败原因存于 error 中	

（2）recv 函数原型

recv（通过 socket 文件描述符从对方接收数据）		
所需头文件	#include <sys/types.h> #include <sys/socket.h>	
函数说明	通过 socket 文件描述符从对方接收数据	
函数原型	ssize_t recv(int s, void *buf, size_t len, int flags)	
函数传入值	s	socket 文件描述符
	len	可接收数据的最大长度
	flags	0：此时的功能同 read，flags 还可以设为以下标志的组合 MSG_OOB：接收带外数据 MSG_PEEK：查看数据标志，返回的数据并不在系统中删除，如果再次调用 recv 函数，会返回相同的数据内容 MSG_DONTWAIT：设置为非阻塞操作 MSG_WAITALL：强迫接收到 len 大小的数据后才返回，除非有错误或信号产生
	buf	接收数据的首地址
函数返回值	成功：实际发送的字节数	
	失败：-1，失败原因存于 error 中	

3. 套接字属性控制函数

系统提供 getsockopt、setsockopt 两个函数获取和修改套接字结构中的一些属性，通过修改这些属性，可以调整套接字的性能，进而调整应用程序的性能。

（1）getsockopt 函数原型

getsockopt（获取套接字的属性）	
所需头文件	#include <sys/types.h> #include <sys/socket.h>
函数说明	获取套接字的属性
函数原型	int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen)

续表

getsockopt（获取套接字的属性）		
函数传入值	s	socket 文件描述符
	level	SOL_SOCKET: 通用套接字选项
		IPPROTO_IP: IP 选项
		IPPROTO_TCP: TCP 选项
	optname	访问的选项名，具体见表 18-3
函数传出值	optlen	optval 的长度
	optval	取得的属性值
函数返回值	成功: 0	
	失败: -1, 失败原因存于 error 中	

表 18-3 列出了套接字及其属性。

表 18-3 套接字属性表

level（级别）	optname（选项名）	说 明	数据类型
SOL_SOCKET	SO_BROADCAST	允许发送广播数据	int
	SO_DEBUG	允许调试	int
	SO_DONTROUTE	不查找路由	int
	SO_ERROR	获得套接字错误	int
	SO_KEEPAIVE	保持连接	int
	SO_LINGER	延迟关闭连接	struct linger
	SO_OOBINLINE	带外数据放入正常数据流	int
	SO_RCVBUF	接收缓冲区大小	int
	SO_SNDBUF	发送缓冲区大小	int
	SO_RCVLOWAT	接收缓冲区下限	int
	SO_SNDLOWAT	发送缓冲区下限	int
	SO_RCVTIMEO	接收超时	struct timeval
	SO_SNDTIMEO	发送超时	struct timeval
	SO_REUSEADDR	允许重用本地地址和端口	int
	SO_TYPE	获得套接字类型	int
	SO_BSDCOMPAT	与 BSD 系统兼容	int
IPPROTO_IP	IP_HDRINCL	在数据包中包含 IP 首部	int
	IP_OPTIONS	IP 首部选项	int
	IP_TOS	服务类型	int
	IP_TTL	生存时间	int
IPPROTO_TCP	TCP_MAXSEG	TCP 最大数据段的大小	int
	CP_NODELAY	不使用 Nagle 算法	int

(2) setsockopt 函数原型

setsockopt（设置套接字的属性）	
所需头文件	#include <sys/types.h> #include <sys/socket.h>

续表

setsockopt (设置套接字的属性)		
函数说明	设置套接字的属性	
函数原型	int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen)	
函数传入值	s	socket 文件描述符
	level	SOL_SOCKET: 通用套接字选项
		IPPROTO_IP: IP 选项
		IPPROTO_TCP: TCP 选项
	optname	设置的选项名, 具体见表 18-3
	optval	设置的属性值
	optlen	optval 的长度
函数返回值	成功: 0	
	失败: -1, 失败原因存于 error 中	

(3) getsockopt、setsockopt 函数举例

sockopt.c 源代码如下:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
int main()
{
    int sockfd,optval,optlen = sizeof(int);
    int sndbuf = 0 ;
    int rcvbuf = 0 ;
    int flag;
    if((sockfd = socket(AF_INET,SOCK_STREAM,0))<0)
    {
        perror("socket") ;
        return -1 ;
    }
    getsockopt(sockfd,SOL_SOCKET,SO_TYPE,&optval,&optlen);
    printf("optval = %d\n",optval);
    optlen = sizeof(sndbuf);
    flag = getsockopt(sockfd,SOL_SOCKET,SO_SNDBUF,&sndbuf,&optlen);
    printf("sndbuf=%d\n",sndbuf) ;
    printf("flag=%d\n",flag) ;
    sndbuf = 51200;
    flag = setsockopt(sockfd,SOL_SOCKET,SO_SNDBUF,&sndbuf, optlen);
    sndbuf=0 ;
    flag = getsockopt(sockfd,SOL_SOCKET,SO_SNDBUF,&sndbuf,&optlen);
    printf("sndbuf=%d\n",sndbuf) ;
    printf("flag=%d\n",flag) ;
    close(sockfd);
    return 0 ;
}
```

编译 gcc sockopt.c -o sockopt。

执行 ./sockopt, 执行结果如下:

```
optval = 1
sndbuf=16384
flag=0
sndbuf=102400
flag=0
```

4. UDP 读写函数

UDP 套接字是无连接协议，必须使用 `sendto` 函数发送数据，使用 `recvfrom` 函数接收数据，且发送时需指明目的地址。`sendto` 函数与 `send` 的功能基本相同，`recvfrom` 与 `recv` 的功能基本相同，只是 `sendto` 函数和 `recvfrom` 函数参数中都带有对方的地址信息，这两个函数是专门为 UDP 协议提供的。

(1) sendto 函数原型

sendto（通过 socket 文件描述符发送数据到对方）		
所需头文件	#include <sys/types.h> #include <sys/socket.h>	
函数说明	通过 socket 文件描述符发送数据到对方，用于 UDP 协议	
函数原型	ssize_t sendto(int s, const void *buf, size_t len, int flags, const struct sockaddr *to, socklen_t tolen)	
函数传入值	s	socket 文件描述符
	buf	发送数据的首地址
	len	发送数据的长度
	flags	0：以默认方式发送数据，flags 还可以设为以下标志的组合 MSG_OOB：发送带外数据 MSG_DONTROUTE：告诉 IP 协议目的主机在本地网络，没有必要查找路由表 MSG_DONTWAIT：设置为非阻塞操作 MSG_NOSIGNAL：表示发送动作不愿被 SIGPIPE 信号中断
	to	存放目的主机 IP 地址和端口信息
	tolen	to 的长度，可设置为 sizeof(struct sockaddr)
函数返回值	成功：实际发送的字节数	
	失败：-1，失败原因存于 error 中	

(2) recvfrom 函数

recv（通过 socket 文件描述符从对方接收数据）		
所需头文件	#include <sys/types.h> #include <sys/socket.h>	
函数说明	通过 socket 文件描述符从对方接收数据，用于 UDP 协议	
函数原型	ssize_t recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen)	
函数传入值	s	socket 文件描述符
	len	可接收数据的最大长度

续表

recv（通过 socket 文件描述符从对方接收数据）		
函数传入值	flags	0：以默认方式接收数据，flags 还可以设为以下标志的组合 MSG_OOB：接收带外数据 MSG_PEEK：查看数据标志，返回的数据并不在系统中删除，如果再次调用 recv 函数，会返回相同的数据内容 MSG_DONTWAIT：设置为非阻塞操作 MSG_WAITALL：强迫接收到 len 大小的数据后才返回，除非有错误或信号产生
	fromlen	from 的长度，可设置为 sizeof(struct sockaddr)
函数传出值	buf	接收数据的首地址
	from	存放发送方的 IP 地址和端口
函数返回值	成功：实际发送的字节数	
	失败：-1，失败原因存于 error 中	

18.2 TCP 套接字编程

18.2.1 TCP 套接字编程模型

TCP 套接字编程经常使用在客户端/服务器编程模型（简称 C/S 模型）中，C/S 模型根据复杂度，可分为简单的客户端/服务器模型和复杂的客户端/服务器模型。简单的客户端/服务器模型是一对一关系，即一个服务器端某一段时间内只对应处理一个客户端的请求，迭代服务器模型属于此模型。复杂的客户端/服务器模型是一对多关系，即一个服务器端某一段时间内对应处理多个客户端的请求，并发服务器模型属于此模型。迭代服务器模型和并发服务器模型是 socket 编程中最常使用的两种编程模型，图 18-5 画出了两种模型服务端的处理流程。

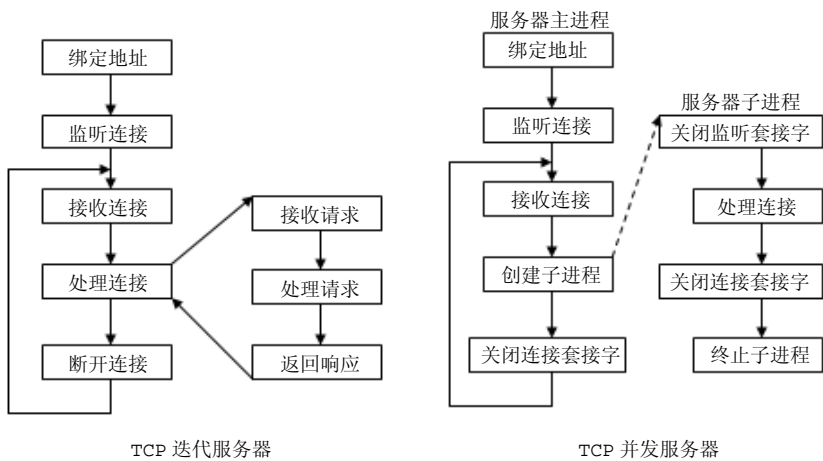


图 18-5 迭代服务器和并发服务器处理流程图

1. TCP 套接字编程模型图

图 18-6 是 TCP 套接字编程模型图，该图演示了客户端与服务器端的编程模型和流程。此模

型不仅适合迭代服务器，也适合并发服务器，两者实现的流程类似，只不过并发服务器接收客户请求（accept）后会用 fork 调用子进程，由子进程处理客户端的请求。

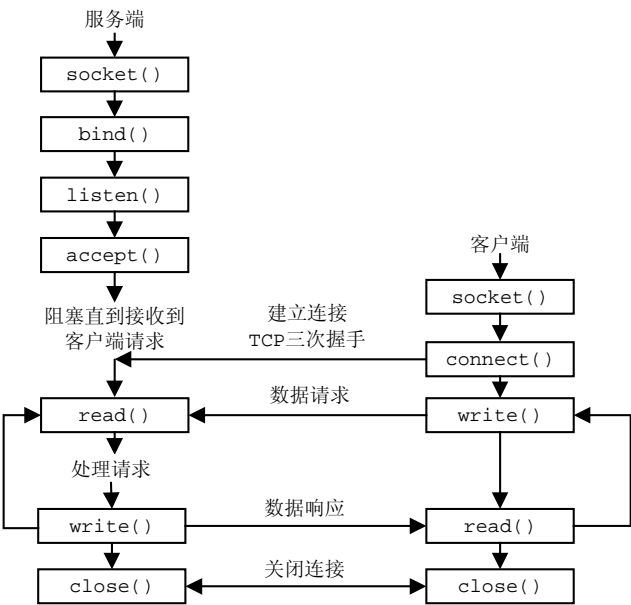


图 18-6 TCP 套接字编程模型图

2. TCP 编程流程说明

(1) 服务器端编程流程

TCP 服务器端编程流程如下：

- ① 创建套接字。
- ② 绑定套接字。
- ③ 设置套接字为监听模式，进入被动接受连接状态。
- ④ 接受请求，建立连接。
- ⑤ 读写数据。
- ⑥ 终止连接。

(2) TCP 客户端编程流程

TCP 客户端编程流程如下：

- ① 创建套接字。
- ② 与远程服务器建立连接。
- ③ 读写数据。
- ④ 终止连接。

（3）TCP 服务器的异常情况

TCP 服务器有三种异常情况，分别为服务器主机崩溃、服务器主机崩溃后重启、服务器主机关机。

在服务器主机崩溃的情况下，已有的网络连接上发不出任何内容。此时应用程序发出数据后，会阻塞于套接字的读取回应。由于服务器主机崩溃，此时客户 TCP 会持续重传数据分节，试图从服务器接收一个 ACK，重传 12 次（源自 Berkeley 的实现）后，客户 TCP 最终选择放弃，返回给应用程序的错误代码为 ETIMEDOUT；或者是因为中间路由器判定服务器主机不可达，则返回一个目的地不可达的 ICMP 消息响应，其错误代码为 EHOSTUNREACH 或 ENETUNREACH。需要说明的是，通过设置套接字选项可以更改 TCP 持续重传等待的超时时间。

在服务器主机崩溃后重启的情况下，如果客户在主机崩溃重启前不主动发送数据，那么客户不会知道服务器已崩溃。在服务器重启后，客户向服务器发送一个数据分节。由于服务器重启后丢失了以前的连接信息（尽管在服务端口上有进程监听，但连接套接字所在的端口无进程等待），因此，导致服务器主机的 TCP 响应 RST，当客户 TCP 收到 RST 时，就向客户返回错误 ECONNRESET。如果客户对服务器的崩溃情况很关心，即使客户不主动发送数据也这样，这需要进行相关设置（如设置套接口选项 SO_KEEPALIVE 或某些客户端/服务器心跳函数）。

当服务器主机关机的情况下，由于 init 进程给所有运行的进程发信号 SIGTERM，这时服务器程序可以捕获该信号，并在信号处理程序中正常关闭网络连接。如果服务器程序忽略了 SIGTERM 信号，则 init 进程会等待一段固定的时间（通常是 5~20s），然后给所有还在运行的程序发信号 SIGKILL。服务器将由信号 SIGKILL 终止，其终止时，所有打开的描述字被关闭，这导致向客户发送 FIN 分节，客户收到 FIN 分节后，能推断出服务器将终止服务。

3. 读写函数的封装

（1）网络数据读写说明

在网络程序中，向套接字文件描述符写数据时有以下两种可能。

① write 的返回值大于 0，表示写了部分或者全部的数据。

② 返回的值小于 0，此时写出现了错误，需要根据错误类型来处理。如果错误号为 EINTR，则为中断引起的，可以忽略进行继续写操作；如果是其他错误号，表示网络连接出现了问题（可能对方关闭了连接），则需报错退出。

与向套接字文件描述符写数据一样，读数据也有两种可能。

① read 的返回值大于 0，表示读了部分或者全部的数据。

② 返回的值小于 0，此时读出现了错误，需要根据错误类型来处理。如果错误号为 EINTR，则为中断引起的，可以忽略进行继续读操作；如果是其他错误号，表示网络连接出现了问题，则需报错退出。

（2）读写函数的封装

为了使读写函数更加健壮，更加易用，需要对读写函数进行封装。tcpio.c 源代码中 readn

函数对 read 函数进行了封装，writen 函数对 write 进行了封装。

tcpio.c 源代码如下：

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
int readn(int fd,void *buffer,int length)
{
    int bytes_left;
    int bytes_read;
    char *ptr;
    bytes_left=length;
    ptr=buffer ;
    while(bytes_left>0)
    {
        bytes_read=read(fd,ptr,bytes_left);
        if(bytes_read<0)
        {
            if(errno==EINTR)
                bytes_read=0;
            else
                return(-1);
        }
        else if(bytes_read==0)
            break;
        bytes_left-=bytes_read;
        ptr+=bytes_read;
    }
    return(length-bytes_left);
}
int writen(int fd,void *buffer,int length)
{
    int bytes_left;
    int written_bytes;
    char *ptr;
    ptr=buffer;
    bytes_left=length;
    while(bytes_left>0)
    {
        written_bytes=write(fd,ptr,bytes_left);
        if(written_bytes<0)
        {
            if(errno==EINTR) /* 错误由中断引起，可以继续写*/
                written_bytes=0;
            else /*其他错误，报错退出*/
                return(-1);
        }
        bytes_left-=written_bytes;
        ptr+=written_bytes;
    }
    return(0);
}
```

18.2.2 迭代服务器编程

下列代码实现的是典型的迭代服务器框架，服务器端的功能是为客户端提供日期服务。

(1) 服务器端代码

dayserv.c 源代码如下：

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <time.h>
#include <unistd.h>
#define MAX_BUFFER      128
#define DAYTIME_SERVER_PORT 8001
int main ( void )
{
    int serverFd, connectionFd;
    struct sockaddr_in servaddr;
    char timebuffer[MAX_BUFFER+1];
    time_t currentTime;
    serverFd = socket(AF_INET, SOCK_STREAM, 0);
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(DAYTIME_SERVER_PORT);
    bind(serverFd, (struct sockaddr *)&servaddr, sizeof(servaddr));
    listen(serverFd, 5);
    while ( 1 ) {
        connectionFd = accept(serverFd, (struct sockaddr *)NULL, NULL);
        if (connectionFd >= 0) {
            currentTime = time(NULL);
            snprintf(timebuffer, MAX_BUFFER, "%s\n", ctime(&currentTime));
            write(connectionFd, timebuffer, strlen(timebuffer));
            close(connectionFd);
        }
    }
    return 0 ;
}
```

(2) 客户端代码

daycli.c 源代码如下：

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <time.h>
#define MAX_BUFFER      128
#define DAYTIME_SERVER_PORT 8001
int main ( int argc, char *argv[] )
```

```
{
    int connectionFd, in;
    struct sockaddr_in servaddr;
    char timebuffer[MAX_BUFFER+1];
    if (argc != 2) {
        fprintf(stderr, "Usage: ./daycli IP\n");
        return -1;
    }
    connectionFd = socket(AF_INET, SOCK_STREAM, 0);
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(DAYTIME_SERVER_PORT);
    servaddr.sin_addr.s_addr = inet_addr(argv[1]);
    connect(connectionFd, (struct sockaddr *)&servaddr, sizeof(servaddr));
    while ( (in = read(connectionFd, timebuffer, MAX_BUFFER)) > 0) {
        timebuffer[in] = 0;
        printf("\n%s", timebuffer);
    }
    close(connectionFd);
    return 0;
}
```

(3) 编译与执行

编译服务端代码 `gcc dayserv.c -o dayserv`。

编译客户端代码 `gcc daycli.c -o daycli`。

执行 `./dayserv &`, `./daycli 127.0.0.1`, 执行结果如下:

Fri Jan 16 14:20:21 2009

18.2.3 并发服务器编程

并发服务器是 `socket` 应用编程中最常见的应用模型。并发服务器模型根据连接方式, 可分为长连接和短连接, 长连接是指通信双方建立连接后一直保持连接, 然后一直用此连接进行读写操作; 短连接是指通信双方每一次交易过程都要建立连接和关闭连接。并发服务器模型根据处理方式, 可分为同步方式和异步方式, 同步是客户端发送请求给服务器等待服务器返回处理结果; 异步是指客户端发送请求给服务器, 不等待服务器返回处理结果, 而直接去完成其他的流程, 对于处理结果, 客户端可以事后查询或让服务器进行主动通知。

1. 并发服务器编程注意事项

进程是一个程序的一次运行过程, 它是一个动态实体, 是独立的任务, 它拥有独立的地址空间、执行堆栈、文件描述符等。每个进程拥有独立的地址空间, 在进程不存在父子关系的情况下, 互不影响。

进程的终止存在两种可能: 父进程先于子进程终止 (由 `init` 进程领养), 子进程先于主进程终止。对于后者, 系统内核为子进程保留一定的状态信息 (进程 ID、终止状态、CPU 时间等), 并向其父进程发送 `SIGCHLD` 信号。当父进程调用 `wait` 或 `waitpid` 函数时, 将获取这些信息,

获取后内核将对僵尸进程进行清理。如果父进程设置了忽略 SIGCHLD 信号或对 SIGCHLD 信号提供了处理函数，即使不调用 wait 或 waitpid 函数，内核也会清理僵尸进程。

父进程调用 wait 函数处理子进程退出信息时，会存在下面所述的问题。在有多个子进程的情况下，wait 函数只等待最先到达的子进程的终止信息。图 18-7 父进程有三个子进程，由于 SIGCHLD 信号不排队，在 SIGCHLD 信号同时到来后，父进程的 wait 函数只执行一次，这样将留下两个“僵尸进程”，使用 waitpid 函数并设置 WNOHANG 选项可以解决这个问题。

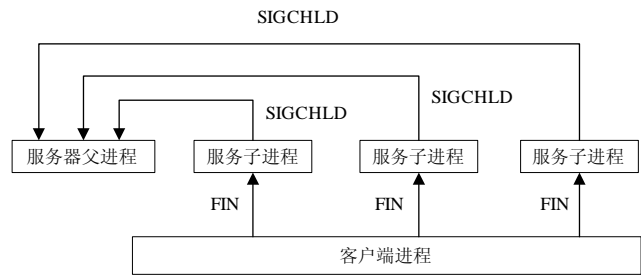


图 18-7 多进程信号图

综上所述，在多线程并发的情况下，防止子进程变成僵尸进程的常见方法有如下两种。

- ① 父进程调用 `signal(SIGCHLD,SIG_IGN)` 对子进程退出信号进行忽略，或者把 SIG_IGN 替换为其他处理函数，设置对 SIGCHLD 信号的处理。
- ② 父进程调用 `waitpid(-1,NULL,WNOHANG)` 对所有的子进程 SIGCHLD 信号进行处理。

2. 并发服务器文件描述符变化图

图 18-8~图 18-11 画出了并发服务器文件描述符的变化流程图。其中，listenfd 为服务端的 socket 监听文件描述符，connfd 为 accept 函数返回的 socket 连接文件描述符。

服务器调用 accept 函数时，客户与服务器文件描述符如图 18-8 所示。

服务器调用 accept 函数后，客户与服务器文件描述符如图 18-9 所示。

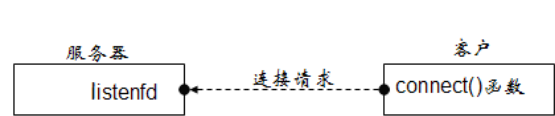


图 18-8 调用 accept 函数时套接字描述符图

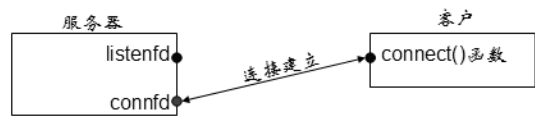


图 18-9 调用 accept 函数后套接字描述符图

服务器调用 fork 函数后，客户与服务器文件描述符如图 18-10 所示。

服务器端父进程关闭连接套接字，子进程关闭监听套接字，客户与服务器文件描述符状况如图 18-11 所示。

在这里强调的是，并发服务器 fork 后父进程一定要关闭子进程连接套接字；而子进程要关闭父进程监听套接字，以免误操作。

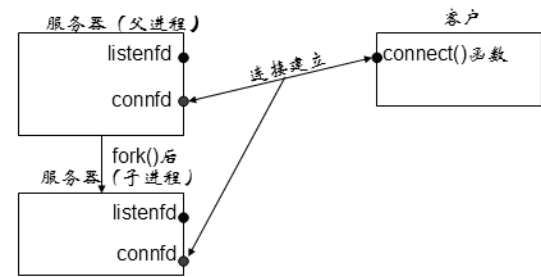


图 18-10 调用 fork 函数后套接字描述符图

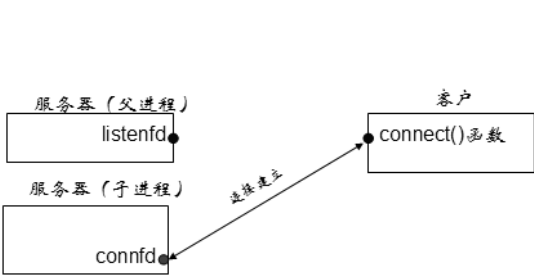


图 18-11 并发服务器最终连接图

3. TCP 并发服务器代码实现

(1) 并发服务器处理流程

并发服务器处理流程如下：

- ① 客户端首先发起连接。
- ② 服务器端进程 `accept` 打开一个新的连接套接字与客户端进行连接，`accept` 在一个 `while (1)` 循环内等待客户端的连接。
- ③ 服务器端 `fork` 一个子进程，同时父进程 `close`（关闭）子进程连接套接字，循环等待下一进程。
- ④ 服务器端子进程 `close` 父进程监听套接字，并用连接套接字保持与客户端的连接，客户发送数据到服务器端，然后阻塞等待服务端返回。
- ⑤ 子进程接收数据，进行业务处理，然后发送数据给客户端。
- ⑥ 子进程关闭连接，然后退出。

(2) 程序报文协议说明

该程序报文协议模式为常见的行业应用软件协议模式，其具体说明如下：

发送和接收报文协议：8 位报文长度（不包含本身）+6 位交易码+报文内容，其中，交易码标识该交易的类型。

实际应用中，服务器端进程根据 6 位交易码调度不同的应用服务。

(3) 并发服务器服务端代码

`tcpsrv.c` 源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
```

```

#include <netinet/in.h>
#include <netinet/tcp.h>
#include <errno.h>
#include <sys/wait.h>
#define MY_PORT 10000
extern int readn(int fd,void *buffer,int length) ;
extern int writen(int fd,void *buffer,int length) ;
int main(int argc ,char **argv)
{
    int listen_fd,accept_fd;
    struct sockaddr_in server_addr;
    struct sockaddr_in cli_addr;
    int n;
    int cliaddr_len ;
    char buffer[1024];
    char data[1024] ;
    long length ;
    int nbytes ;
    if((listen_fd=socket(AF_INET,SOCK_STREAM,0))<0)
    {
        printf("socket Error:%s\n",strerror(errno));
        return -1;
    }
    memset(&server_addr,0x00, sizeof(struct sockaddr_in));
    memset(&cli_addr,0x00, sizeof(struct sockaddr_in));
    server_addr.sin_family=AF_INET;
    server_addr.sin_port=htons(MY_PORT);
    server_addr.sin_addr.s_addr=htonl(INADDR_ANY);
    n=1;
    /*服务器如果终止,可以第二次快速启动而不用等待一段时间 */
    setsockopt(listen_fd,SOL_SOCKET,SO_REUSEADDR,&n,sizeof(int));
    if(bind(listen_fd,(struct sockaddr *)&server_addr,sizeof(server_addr))<0)
    {
        printf("Bind Error:%s\n",strerror(errno));
        return -1;
    }
    listen(listen_fd,5);
    while(1)
    {
        cliaddr_len= sizeof( cli_addr ) ;
        accept_fd=accept(listen_fd, (struct sockaddr *)&cli_addr, &cliaddr_len );
        if((accept_fd<0)&&(errno==EINTR))
            continue;
        else if(accept_fd<0)
        {
            printf("Accept Error:%s\n",strerror(errno));
            continue;
        }
        if((n=fork())==0)
        {
            /* 子进程处理客户端的连接 */
            fprintf(stdout,"listen_fd:%d accept_fd:%d\n",listen_fd, accept_fd);
            close(listen_fd);
            memset(buffer, 0x00, sizeof(buffer)) ;
            memset(data, 0x00, sizeof(data));
            if((nbytes=readn(accept_fd, data, 8 ))==-1)

```

```

    {
        fprintf(stderr, "Read Error:%s\n", strerror(errno));
        fprintf(stderr, "data:%s\n", data );
        close(accept_fd);
        return -1;
    }
    fprintf(stdout, "data:%s,nbytes=%d\n", data, nbytes );
    data[nbytes]='\0' ;
    length=atol(data) ;
    fprintf(stdout, "data:%s,nbytes=%d\n", data, nbytes );
    if((nbytes=readn(accept_fd, data, length ))==-1)
    {
        fprintf(stderr, "Read Error:%s\n", strerror(errno));
        close(accept_fd);
        return -1;
    }
    data[nbytes]='\0' ;
    fprintf(stdout, "data:%s,nbytes=%d\n", data, nbytes );
    if( strncmp(data, "000000", 6 )==0 )
    {
        strcpy(buffer, "I am sorry! who am I? I don't know also.") ;
        length=strlen(buffer) ;
        sprintf(data, "%08ld%6.6s%s", (length+6), "000000", buffer ) ;
        if((nbytes=writen(accept_fd, data, (length+6+8)))== -1)
        {
            fprintf(stderr, "Read Error:%s\n", strerror(errno));
            close(accept_fd);
            return -1;
        }
        fprintf(stdout, "data:%s\n", data );
    }else{
        /*非 000000 交易请求为非法，沉默是最好的回答*/
    }
    close(accept_fd);
    return 0;
}
else if(n<0)
    printf("Fork Error:%s\n\a",strerror(errno));
close(accept_fd);
while(waitpid(-1,NULL,WNOHANG) > 0); /* clean up child processes */
}
}

```

(4) 客户端代码

tcpcli.c 源代码如下:

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>

```

```

#include <errno.h>
extern int readn(int fd,void *buffer,int length) ;
extern int writen(int fd,void *buffer,int length) ;
int main(int argc, char *argv[])
{
    int sockfd;
    char buffer[1024];
    char data[1024];
    long length ;
    struct sockaddr_in server_addr;
    struct hostent *host;
    int portnumber,nbytes;
    if(argc!=3)
    {
        fprintf(stderr,"Usage:%s hostname portnumber\n",argv[0]);
        return -1;
    }
    if((host=gethostbyname(argv[1]))==NULL)
    {
        fprintf(stderr,"Gethostname error\n");
        return -1;
    }
    if((portnumber=atoi(argv[2]))<0)
    {
        fprintf(stderr,"Usage:%s hostname portnumber\n",argv[0]);
        return -1;
    }
    /* 客户程序开始建立 sockfd 描述符 */
    if((sockfd=socket(AF_INET,SOCK_STREAM,0))== -1)
    {
        fprintf(stderr,"socket Error:%s\n",strerror(errno));
        return -1;
    }
    /* 客户程序填充服务端的地址端口信息 */
    bzero(&server_addr,sizeof(server_addr));
    server_addr.sin_family=AF_INET;
    server_addr.sin_port=htons(portnumber);
    server_addr.sin_addr= *((struct in_addr *)host->h_addr);
    /* 客户程序发起连接请求 */
    if(connect(sockfd,(struct sockaddr *)&server_addr,sizeof(struct sockaddr)
        )== -1)
    {
        fprintf(stderr,"Connect Error:%s\n",strerror(errno));
        return -1;
    }
    memset(buffer, 0x00, sizeof(buffer)) ;
    memset(data, 0x00, sizeof(data));
    strcpy(buffer, "Hello! who are you? could you tell me?" ) ;
    length=strlen(buffer) ;
    /**000000 为假设的交易码***/
    sprintf(data,"%08ld%6s%s", (length+6),"000000", buffer ) ;
    length=length+8+6 ;
    if((nbytes=writen(sockfd,data, length ))== -1)
    {
        fprintf(stderr,"Read Error:%s\n",strerror(errno));
    }
}

```



```

        close(sockfd);
        return -1;
    }
    printf("I have send:%s\n", data+8);
    if((nbytes=readn(sockfd, data, 8 ))==-1)
    {
        fprintf(stderr,"Read Error:%s\n",strerror(errno));
        close(sockfd);
        return -1;
    }
    data[nbytes]='\0' ;
    length=atol(data) ;
    if((nbytes=readn(sockfd, data, length ))==-1)
    {
        fprintf(stderr,"Read Error:%s\n",strerror(errno));
        close(sockfd);
        return -1;
    }
    data[nbytes]='\0' ;
    printf("I have received:%s\n", data);
    close(sockfd);
    return 0;
}

```

(5) 编译与执行

编译 `gcc tcpsrv.c tcpio.c -o tcpsrv。`

编译 `gcc tcpcli.c tcpio.c -o tcpcli。`

在一个界面中启动服务端进程 `./tcpsrv。`

在另一个界面中执行 `./tcpcli 127.0.0.1 10000`，执行结果如下：

```

I have send:000000Hello! who are you? could you tell me?
I have received:000000I am sorry! who am I? I don't know also.

```

18.3 TCP 文件服务器项目案例

1. 文件服务器案例介绍

文件服务器的意义在于简化了 C/S 编程的文件传输。客户端接收报文后，直接调用接收文件函数接口从文件服务器上取回文件。客户端需要上传文件时，直接调用发送文件函数接口上传文件到文件服务器。

本例的文件服务器是实际应用中的代码改造而成的，代码中 `alarm` 函数的作用是完成进程读写的超时控制。该文件服务端还存在报错处理方面的不足，请读者自行优化。文件服务器服务端和客户端执行码可以放在两个机器上或同一台机器的两个用户下。使用此文件服务器时，需要在服务端和客户端设置文件存放路径的环境变量，设置方法如下：

```
FILEDIR=$HOME/print/;export FILEDIR
```

文件服务器首次通信的报文头格式为：8 位文件长度+50 位文件名+2 位控制位。其中，2 位控制位 00 表示上传文件，10 表示下载文件。

2. 文件服务器代码实现

(1) 公共头文件

atp_file.h 头文件源代码如下：

```
#ifndef    _AP_FILE_H
#define    _AP_FILE_H
#include <arpa/inet.h>
#include <errno.h>
#include <fcntl.h>
#include <limits.h>
#include <netdb.h>
#include <netinet/in.h>
#include <poll.h>
#include <setjmp.h>
#include <signal.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/sem.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <time.h>
#include <unistd.h>
#define MAXSIG 32
#define TIMEOUT 30
#define _FAIL_ (-1)
#define mylog(msg) { syslog("atp","log",__FILE__,__LINE__,"%s",msg); }
#define errlog(msg) { syslog("atp","err",__FILE__,__LINE__,"%s",msg); }
#define debug(msg) { syslog("atp","debug",__FILE__,__LINE__,"%s",msg); }
int sys_return ;
struct sockaddr_in SIN;
char BEGINTIME[20] ;
char ENDTIME[20] ;
char LOCALIP[20] ;
int LOCALPORT ;
int Creat_S ( char *Hostname, int Port );
int Server_S( int iSOCKETNO, void(*proc)(int));
int Read_S ( int iSOCKETNO,char *sss, long Length, int time );
int Send_S ( int iSOCKETNO,char *sss, long Length, int time );
void timeout() ;
int stringpack( char * str );
char *_strl(char *fmt,...) ;

int recvfilesock(int filesock, char *recvfile,long filelength ) ;
```

```

int sendfilesock(int filesock, char *sendfile) ;
int filesend(char *ip, int port, char *filename) ;
int filerecv(char *ip, int port, char *filename) ;

#endif

```

(2) syslog 日志函数

syslog 日志函数见 8.2.2 节中的 syslog.c 文件。

(3) 网络通信库文件

netproc.c 源代码如下:

```

#include "atp_file.h"
jmp_buf env ;
void timeout() ;
void timeout()
{
    sigset(SIGALRM , SIG_IGN) ;
    longjmp(env, 1 ) ;
}
/****下面两个函数是获取一个进程的所有占用时间****/
int Set_ptime()
{
    long s_time ;
    struct tm *ts;
    s_time = time( ( long * ) 0 ) ;
    ts = localtime( &s_time );
    memset( BEGINTIME , 0x00 , sizeof( BEGINTIME ) ) ;
    sprintf( BEGINTIME , "%02d:%02d:%02d", ts->tm_hour, ts->tm_min, ts->tm_sec);
    return 0 ;
}
int Get_ptime()
{
    long s_time ;
    struct tm *ts;
    s_time = time( ( long * ) 0 ) ;
    ts = localtime( &s_time );
    memset( ENDTIME , 0x00 , sizeof( ENDTIME ) ) ;
    sprintf( ENDTIME , "%02d:%02d:%02d", ts->tm_hour, ts->tm_min, ts->tm_sec);
    return 0 ;
}
int Creat_S ( char *Hostname, int Port )
{
    int iSOCKETNO;
    struct hostent *hostp;
    if(Hostname==NULL)
    {
        return(-1);
    }
    if((hostp=gethostbyname(Hostname))==NULL)
    {
        errlog( _strl("Hostname:%s is not exist",Hostname) );
        exit(-1);
    }
}

```

```

}
if((iSOCKETNO=socket(AF_INET,SOCK_STREAM,0))<0)
{
    perror(" socket error");
    exit(-1);
}
SIN.sin_family=AF_INET;
SIN.sin_port=htons(Port);
bcopy(hostp->h_addr_list[0],&SIN.sin_addr,hostp->h_length);
return(iSOCKETNO);
}
int Server_S( int iSOCKETNO, void(*proc)(int))
{
    unsigned int flll=0;
    int n,ipid , child_id;
    int vret ;
    int on=1 ;
    setsockopt(iSOCKETNO , SOL_SOCKET , SO_REUSEADDR , (char *)&on , sizeof(on) );
    for(n=1;n<MAXSIG;n++)
        signal(n,SIG_IGN);
    signal(SIGINT,SIG_DFL);
    if(bind(iSOCKETNO,(struct sockaddr *)&SIN,sizeof(struct sockaddr))<0)
    {
        perror(" bind ....");
        exit(-1);
    }
    if(listen(iSOCKETNO,5)<0)
    {
        perror("listen...");
        exit(-1);
    }
    child_id = fork() ;
    if(child_id > 0)                /* 父进程退出          */
        exit(0);
    setpgpr();                    /* 设置进程的属组      */
    signal(SIGHUP,SIG_IGN);       /* 忽略 hangup 信号    */
    child_id = fork();            /* 产生第二个子进程    */
    if(child_id < 0){
        printf("不能产生第二个子进程 ...\n");
        exit(1);
    }
    if(child_id > 0)                /* 父进程退出          */
        exit(0);
    errno = 0;                    /* 清除错误代码        */
    vret = chdir("/usr/tmp");     /* 设置默认目录        */
    umask(0);                     /* 清除掩码            */
    signal(SIGCLD,SIG_IGN);       /* 忽略子进程死的信号  */
    while(1)
    {
        if((n=accept(iSOCKETNO,(struct sockaddr *)&SIN,&flll))<0)
        {
            errlog( _strl("pid[%d]no[%d]number[%d]",getpid(),iSOCKETNO,n) );
            perror("accept...");
            getchar();
            continue;
        }
    }
}

```

```

    }

    if((ipid=fork())<0)
    {
        perror( " fork error ");
        exit (-1);
    }
    if( ipid > 0 )
    {
        close ( n ) ;
        continue ;
    }
    if(ipid==0)
    {
        close(iSOCKETNO);
        iSOCKETNO=n;
        sys_return = _FAIL_ ;
        Set_ptime() ;
        proc(iSOCKETNO);
        if ( sys_return )
        {
            errlog( _strl("running process[%d] 开始时间[%s] 结束时间[%s]",
getpid() , BEGINTIME , ENDTIME) ) ;
            close(iSOCKETNO);
            exit(0);
        }
        close(iSOCKETNO);
        Get_ptime() ;
        mylog( _strl("pid=[%d][%s-%s]FINISH",getpid(),BEGINTIME,ENDTIME) );
        exit(0);
    }
}

}

int Client_S( int iSOCKETNO , void ( *proc)(int))
{
    int n;
    for(n=1;n<MAXSIG;n++)
        signal(n,SIG_IGN);
    signal(SIGINT,SIG_DFL);
    if(connect(iSOCKETNO,(struct sockaddr *)&SIN , sizeof(struct sockaddr_in))<0)
    {
        perror ( " error in connect ");
        exit(-1);
    }
    proc(iSOCKETNO);
    close(iSOCKETNO);
    return(0);
}

int Read_S ( int iSOCKETNO ,char *sss , long Length,int time )
{
    long kk=0;
    long left ;
    memset(sss,0x0,Length+1);

    if ( setjmp(env) !=0 )
    {

```

```

        errlog( " timeout !!! " );
        return(100);
    }
    sigset( SIGALRM ,timeout);
    alarm(time);

    left = Length ;
    while( left > 0 )
    {
        kk=read ( iSOCKETNO , sss , left ) ;
        if( errno == ENOSR )
        {
            errno = 0 ;
            continue ;
        }

        if ( kk < 0 )
        {
            if ( errno == EINTR )
            {
                errno = 0 ;
                continue ;
            }
            break ;
        }
        if ( kk == 0 )
            break ;
        left -= kk ;
        sss += kk ;

        kk=0 ;
    }
    alarm(0);
    sigset(SIGALRM , SIG_IGN);
    if(left != 0 )
    {
        close ( iSOCKETNO ) ;
        errlog( _strl(" 读数据错 [%ld] ==>%ld ",left , kk) );
        return(-1) ;
    }
    return(0);
}

int Send_S ( int iSOCKETNO ,char *sss , long Length , int  time)
{
    long kk=0;
    long left ;

    if ( setjmp(env) !=0 )
    {
        errlog( " timeout !!! " );
        return(100);
    }

    sigset( SIGALRM , timeout);
    alarm(time);
    left = Length ;

```

```

while ( left > 0 )
{
    kk = 0 ;
    kk = write ( iSOCKETNO , sss , left ) ;
    if ( errno == EINTR )
    {
        if ( kk < 0 )
        {
            errno = 0 ;
            continue ;
        }
        break ;
    }
    if ( kk <= 0 )
    {
        break ;
    }

    left -= kk ;
    sss += kk ;

}
alarm(0);
sigset(SIGALRM , SIG_IGN);
if(left!=0)
{
    close ( iSOCKETNO ) ;
    errlog( _strl("发送数据 error size[%ld] ==>[%ld] ",left ,kk ) );
    return -1;
}
return(0);
}
/**** 与第三方或者主机建立连接, 返回 socket_id *****/
int Client_socket( char *ip , int port )
{
    struct hostent *hostp2;
    struct sockaddr_in SIN2;
    int client_is ;
    char acErrMsg[512] ;

    sprintf( acErrMsg , "ipaddress ==>%s port ==>%d\n " , ip , port ) ;
    errlog( acErrMsg ) ;
    if((hostp2=gethostbyname(ip))==NULL)
    {
        sprintf ( acErrMsg , "Hostname : %s is not exist !\n", ip);
        errlog( acErrMsg ) ;
        return -1;
    }
    if((client_is=socket(AF_INET,SOCK_STREAM,0))<0)
    {
        sprintf( acErrMsg , " open client socket error ");
        errlog( acErrMsg ) ;
        return -1;
    }
    SIN2.sin_family=AF_INET;

```

```

SIN2.sin_port=htons(port);
bcopy(hostp2->h_addr,&SIN2.sin_addr,hostp2->h_length);
if(connect(client_is,(struct sockaddr *)&SIN2 , sizeof(struct sockaddr_in))<0)
{
    sprintf( acErrMsg , " connect to client error [%d] " , errno );
    errlog( acErrMsg ) ;
    return -1;
}
return client_is ;
}

```

(4) 文件接口库

filesock.c 源代码如下:

```

#include "atp_file.h"
#define LEN 512
long getFileSize( char *filename )
{
    struct stat f_stat;

    if ( stat( filename, &f_stat ) == -1 ) {
        return -1;
    }
    return f_stat.st_size;
}
char *_strl(char *fmt,...)
{
    static char tmpbuf[2048];
    va_list ap;
    va_start(ap,fmt);
    vsprintf(tmpbuf,fmt,ap);
    va_end(ap);
    tmpbuf[sizeof(tmpbuf)-1]=0;
    return tmpbuf;
}
/* 去掉字符串的空格 */
int stringpack( char *str )
{
    char * pp , * p1 , buff[4096] ;

    pp = str ;
    if( * pp == '\0' ) return 0 ;
    while( * pp == ' ' || * pp == '\t' ) pp ++ ;
    for( p1 = buff ; * pp ; * p1 = * pp , p1 ++ , pp ++ ) ;
    p1 = '\0' ;
    pp = buff + strlen( buff ) - 1 ;
    while( pp >= buff && ( * pp == ' ' || * pp == '\t' ) ) pp -- ;
    if( pp < buff ) buff[0] = '\0' ;
    else * ( pp + 1 ) = '\0' ;
    for( p1 = buff , pp = str ; * p1 ; * pp = * p1 , p1 ++ , pp ++ ) ;
    * pp = '\0' ;
    return 0 ;
}
/*-----
Function Name : recvfilesock

```


Description : 接收文件函数

Input : filesock —— socket 接口值
 recvfile —— 接收文件路径及文件名称
 filelength —— 接收文件的长度

```
-----*/
int rcvfilesock(int filesock, char *recvfile,long filelength )
{
    long len1 = 0 ;
    int file_len = 0 ;
    FILE *fp ;
    char fbuff[512+1] ;

    fp = fopen( recvfile , "w" ) ;
    if( fp ==NULL)
    {
        syslog("tcp","file",__FILE__,__LINE__, "open file error" ) ;
        return _FAIL_ ;
    }

    len1 = filelength ;
    if( len1 > 0 )
    {
        for ( ;len1 > LEN ; len1 -= LEN )
        {
            memset( fbuff , 0x00 , sizeof( fbuff ) ) ;
            if( Read_S ( filesock , fbuff , LEN , TIMEOUT ) )
            {
                syslog("tcp","file",__FILE__,__LINE__, "read file error " ) ;
                fclose( fp ) ;
                return _FAIL_ ;
            }
            file_len = fwrite( fbuff , LEN , 1 ,fp ) ;
        }
        if( len1 > 0 )
        {
            memset( fbuff , 0x00 , sizeof( fbuff ) ) ;

            memset( fbuff , 0x00 , sizeof( fbuff ) ) ;
            if( Read_S ( filesock , fbuff , len1 , TIMEOUT ) )
            {
                syslog("tcp","file",__FILE__,__LINE__, "read file error " ) ;
                fclose( fp ) ;
                return _FAIL_ ;
            }
            file_len = fwrite( fbuff ,len1 , 1 ,fp ) ;
        }
    }
    fclose( fp ) ;
    return 0;
}
/*-----*/
```

Function Name : sendfilesock

Description : 发送文件函数

Input : filesock —— socket 接口值
 sendfile —— 发送文件路径及文件名称

```

-----*/
int sendfilesock(int filesock, char *sendfile)
{
    long filelength = 0 ;
    long len1 = 0 ;
    int file_len = 0 ;
    FILE *fp ;
    char fbuff[512+1] ;

    filelength = getFileSize( sendfile ) ;
    len1 = filelength ;
    memset(fbuff, 0x00, sizeof( fbuff )) ;

    if( ( fp = fopen( sendfile, "rb" ) ) == NULL )
    {
        syslog("tcp","file",__FILE__,__LINE__, _strl("open file error,file=%s",
sendfile) );
        return _FAIL_ ;
    }
    if( len1 > 512 )
    {
        for ( ;len1 > LEN ; len1 -= LEN )
        {
            memset( fbuff , 0x00 , sizeof( fbuff ) ) ;
            file_len = fread( fbuff,1, 512 ,fp );
            if( Send_S( filesock , fbuff , LEN, TIMEOUT ) )
            {
                syslog("tcp","file",__FILE__,__LINE__,_strl("send file error
data[%s]\n",fbuff) );
                fclose( fp ) ;
                return _FAIL_ ;
            }
            syslog("tcp","file",__FILE__,__LINE__, "send data[%s]\n", fbuff ) ;
        }
        if( len1 > 0 )
        {
            memset( fbuff , 0x00 , sizeof( fbuff ) ) ;
            len1 = fread( fbuff,1, 512 ,fp );
            if( Send_S( filesock , fbuff , len1, TIMEOUT ) )
            {
                syslog("tcp","file",__FILE__,__LINE__,_strl("send file error
data[%s]\n",fbuff) );
                fclose( fp ) ;
                return _FAIL_ ;
            }
            syslog("tcp","file",__FILE__,__LINE__, "send data[%s]\n", fbuff ) ;
        }
    }else{
        memset( fbuff , 0x00 , sizeof( fbuff ) ) ;
        len1 = fread( fbuff,1, 512 ,fp );
        if( Send_S( filesock , fbuff , len1, TIMEOUT ) )
        {
            syslog("tcp","file",__FILE__,__LINE__,_strl("send file error data[%s]\n",
fbuff) );
            fclose( fp ) ;
            return _FAIL_ ;
        }
    }
}

```

```

    }
    syslog("tcp","file",__FILE__,__LINE__, "send data[%s]\n", fbuff ) ;
}
fclose( fp ) ;
return 0 ;
}
/*发送文件接口函数,形参为 IP 地址、端口、文件名*/
int filesend(char *ip, int port, char *filename)
{
    char filehead[60+1];
    char pathfile[128] ;
    char *p,*pstart;
    int fileSock=0;
    int len=0;
    int ret=0;
    int filelength=0 ;
    memset(filehead, 0x00, sizeof(filehead)) ;
    fileSock = Client_socket( ip , port ) ;
    if( fileSock <=0 )
    {
        close(fileSock) ;
        syslog("tcp","file",__FILE__,__LINE__, "connect error,ip[%s],port[%d]",
ip,port);
        return -1 ;
    }
    sprintf(pathfile, "%s%s", getenv("FILEDIR"), filename ) ;
    filelength = getFileSize( pathfile ) ;
    sprintf(filehead,"%08d%50.50s%c%c", filelength, filename,'0','0') ;
    printf("filehead=%s\n", filehead) ;
    p = filehead ;
    ret = Send_S( fileSock ,p, 60, TIMEOUT);
    if( ret )
    {
        close(fileSock) ;
        syslog("tcp","file",__FILE__,__LINE__, "发送报文头失败");
        return -1 ;
    }
    ret = sendfilesock(fileSock, pathfile);
    if ( ret < 0 )
    {
        close(fileSock) ;
        syslog("tcp","file",__FILE__,__LINE__, "发送文件失败");
        return -1 ;
    }
    close(fileSock) ;
    return 0 ;
}
int filerecv(char *ip, int port, char *filename)
{
    char filehead[60+1];
    char pathfile[128] ;
    char *p,*pstart;
    int fileSock=0;
    int len=0;
    int ret=0;

```

```

int filelength=0 ;
char buffer[128] ;
memset(filehead, 0x00, sizeof(filehead)) ;
fileSock = Client_socket( ip , port ) ;
if( fileSock <=0 )
{
    close(fileSock) ;
    syslog("tcp","file",__FILE__,__LINE__,"connect error,ip[%s],port[%d]",
ip,port);
    return -1 ;
}
sprintf(pathfile, "%s%s", getenv("FILEDIR"), filename ) ;
filelength = 0 ;
sprintf(filehead,"%08d%50.50s%c%c", filelength, filename,'1','0') ;
p=filehead ;
ret = Send_S( fileSock ,p, 60, TIMEOUT);
if( ret )
{
    close(fileSock) ;
    syslog("tcp","file",__FILE__,__LINE__,"发送报文失败");
    return -1 ;
}
memset(buffer, 0x00, sizeof(buffer)) ;
p=buffer ;
/* 接收 8 B 的报文长度 */
ret = Read_S( fileSock ,p, 8, TIMEOUT);
if( ret )
{
    close(fileSock) ;
    syslog("tcp","file",__FILE__,__LINE__,"接收报文头失败");
    return -1 ;
}
filelength=atol(p) ;
ret = recvfilesock( fileSock, pathfile, filelength ) ;
if ( ret < 0 )
{
    close(fileSock) ;
    syslog("tcp","file",__FILE__,__LINE__,"接收文件失败");
    return -1 ;
}
close(fileSock) ;
return 0 ;
}

```

(5) 文件服务器服务端代码

atpfileseser.c 源文件如下:

```

/*****
*   ProgramName : atpfileseser.c
*   Language    : C
*   OS & Env    : Linux
*   Description  : 文件服务器
*   YYYY/MM/DD  Position      Author      Description
*****/

```

```

#include "atp_file.h"
/*-----
 * Function Name : process()
 * Description   : 文件主服务进程
 * Input        : socket_server -- server socket
 * Output       :
 * Return       :
 *-----*/
static void process( int fileSock )
{
    char filehead[100+1];
    long filelength = 0;
    char filename[50+1];    /***文件名称***/
    char pathfile[128];    /***文件路径全称***/
    char lengthchar[8+1];  /***长度串***/
    char control[2+1];     /***两位控制位***/
    char fbuff[512];
    int ret;
    memset(filehead, 0x00, sizeof(filehead));
    /***接收 60 位报文头***/
    ret = Read_S( fileSock , filehead , 60 ,TIMEOUT );
    if ( ret )
    {
        syslog("tcp","file",__FILE__,__LINE__, "接收文件头出错" );
        close(fileSock);
        return ;
    }
    /***8 位文件长度***/
    memset(lengthchar, 0x00, sizeof(lengthchar));
    strncpy( lengthchar, filehead, 8 );
    stringpack( lengthchar );
    filelength = atol( lengthchar );
    /***50 位文件名称***/
    memset( filename, 0x00, sizeof( filename ) );
    strncpy( filename, filehead+8, 50 );
    stringpack( filename );
    /***2 位控制位***/
    memset( control, 0x00, sizeof( control ) );
    strncpy( control, filehead+58, 2 );
    sprintf(pathfile, "%s%s", getenv("FILEDIR"), filename );
    syslog("tcp","file",__FILE__,__LINE__, "报文头=%s", filehead);
    syslog("tcp","file",__FILE__,__LINE__, "文件长度=%ld", filelength);
    syslog("tcp","file",__FILE__,__LINE__, "文件名称=%s", filename );
    syslog("tcp","file",__FILE__,__LINE__, "控制位=%s", control );
    if(control[0]=='0') /*上传文件*/
    {
        unlink( pathfile );
        ret = recvfilesock(fileSock, pathfile, filelength );
        if ( ret < 0 )
        {
            syslog("tcp","file",__FILE__,__LINE__, "接收报文头失败");
        }
    }
    else if (control[0]=='1') /*下传文件*/

```

```

{
    filelength = getFileSize( pathfile ) ;
    memset(fbuff, 0x00, sizeof( fbuff )) ;
    sprintf(fbuff,"%08ld", filelength ) ;
    if( Send_S( fileSock , fbuff , 8 , TIMEOUT ) )
    {
        syslog("tcp","file",__FILE__,__LINE__, _strl("send data error") );
        return ;
    }
    ret = sendfilesock(fileSock, pathfile);
    if ( ret < 0 )
    {
        syslog("tcp","file",__FILE__,__LINE__, "发送报文失败");
    }
}
else{
    syslog("tcp","file",__FILE__,__LINE__, "上下传控制位不对");
}
sys_return = 0 ;
close(fileSock);
return;
}
int main()
{
    int serverSocket ;
    strcpy(LOCALIP, "192.168.120.130") ;
    LOCALPORT = 9999 ;
    if (( serverSocket = Creat_S( LOCALIP , LOCALPORT ) ) <= 0 )
    {
        errlog( _strl("listen fail[%s:%d]", LOCALIP, LOCALPORT ) );
        return _FAIL_;
    }
    mylog( _strl("LISTEN at %s:%d", LOCALIP , LOCALPORT ) );
    fprintf( stderr, "Daemon running in backgroup ... OK !!!!\n\n" );
    Server_S ( serverSocket , process );
    return -1;
}

```

(6) 发送客户端代码

fsend.c 源代码如下:

```

#include "atp_file.h"
int main(int argc, char *argv[])
{
    char serv_addr[20];
    int port ;
    char filename[50+1] ;

    if ( argc !=4 )
    {
        fprintf( stderr, "\tUsage:\n\t "
                " ./fsend <serv_ip> <serv_port> <file_name>\n" );
        return -1;
    }
    strcpy( serv_addr, argv[1] ) ;
    port = atoi(argv[2]) ;

```

```

sprintf( filename, "%s", argv[3]) ;
if ( filesend(serv_addr, port, filename) )
{
    printf("send file error\n") ;
    return -1 ;
}
printf("send file success\n") ;
return 0 ;
}

```

(7) 接收客户端代码

frecv.c 源代码如下:

```

#include "atp_file.h"
int main(int argc, char *argv[])
{
    char serv_addr[20];
    int port ;
    char filename[50+1] ;

    if ( argc !=4 )
    {
        fprintf( stderr, "\tUsage:\n\t "
            " ./frecv <serv_ip> <serv_port> <file_name>\n" );
        return -1;
    }

    strcpy( serv_addr, argv[1] ) ;
    port = atoi(argv[2]) ;
    sprintf( filename, "%s", argv[3]) ;
    if ( filerecv(serv_addr, port, filename) )
    {
        printf("recv file error\n") ;
        return -1 ;
    }
    printf("recv file success\n") ;
    return 0 ;
}

```

(8) makefile 文件

makefile 文件内容如下:

```

.SUFFIXES: .c .o.c
.c.o:
    cc -O -c -g $(LFLAGS) $*.c

OBJ1=atpfileseser.o syslog.o netproc.c filesock.c
OBJ2=fsend.o syslog.o netproc.c filesock.c
OBJ3=frecv.o syslog.o netproc.c filesock.c

all:atpfileseser fsend frecv

atpfileseser: $(OBJ1)
    gcc -o $@ $(OBJ1) $(LFLAGS)

```

```
fscd: $(OBJ2)
gcc -o $@ $(OBJ2) $(LFLAGS)

frecv: $(OBJ3)
gcc -o $@ $(OBJ3) $(LFLAGS)

c:
rm -f *.o
```

(9) 编译与测试

- ① 在命令行使用 `make all` 命令编译全部执行码。
- ② 设置服务器和客户端 `FILEDIR` 环境变量。
- ③ 启动服务器端侦听进程 `./atpfileserv`。
- ④ 在客户端 `$FILEDIR` 目录下建立 `send.txt` 文件。
- ⑤ 在服务端 `$FILEDIR` 目录下建立 `recv.txt` 文件。
- ⑥ 使用 `./fscd 192.168.120.130 9999 send.txt` 发送文件到文件服务器，其中，`192.168.120.130` 为测试文件服务器 IP 地址。
- ⑦ 使用 `./frecv 192.168.120.130 9999 recv.txt` 从文件服务器上取文件。

18.4 UDP 编程

UDP 协议是非连接非可靠的数据传输，常用在对数据质量要求不高的场合。UDP 服务器通常是非连接的，因而，UDP 服务器进程不需要像 TCP 服务器那样在侦听套接字上接收新建的连接；UDP 只需要在绑定的端口上等待客户机发送来的 UDP 数据报文，并对其进行处理和响应。一个 TCP 服务器进程只有在完成了对某客户机的服务后，才能为其他的客户机提供服务。而 UDP 服务器只是接收数据报文，处理并返回结果。UDP 支持广播和多播，如果要使用广播和多播，必须使用 UDP 套接字。UDP 套接字没有连接的建立和终止过程，UDP 只需要两个分组来交换一个请求和应答。UDP 不适合海量数据的传输。

UDP 编程常使用在三种场合，分别为普通 UDP 服务器编程、UDP 广播、UDP 多播，下面将逐一介绍这三种 UDP 编程模型，并给出范例程序。

18.4.1 普通 UDP 服务器编程

1. 普通 UDP 服务器编程模型

(1) 普通 UDP 服务器端编程流程

普通 UDP 服务器端编程流程如下：

- ① 建立 UDP 套接字。
- ② 绑定套接字到特定的地址。
- ③ 等待并接收客户端信息。
- ④ 处理客户端请求。
- ⑤ 发送信息给客户端。
- ⑥ 关闭套接字。

(2) UDP 客户端流程

UDP 客户端编程流程如下：

- ① 建立 UDP 套接字。
- ② 发送信息给服务器。
- ③ 接收来自服务器的信息。
- ④ 关闭套接字。

(3) 普通 UDP 服务器编程模型

图 18-12 画出了普通 UDP 服务器端和客户端的编程模型。

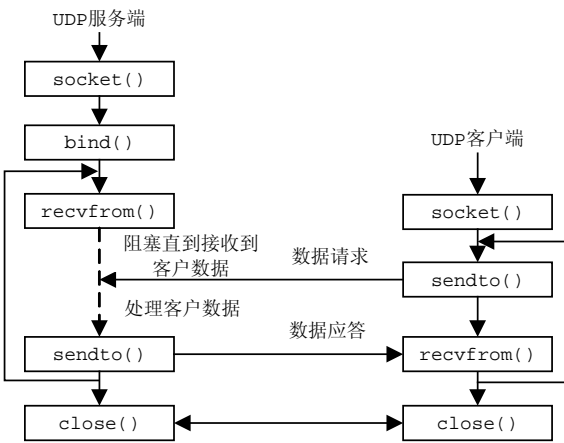


图 18-12 普通 UDP 服务器编程模型图

2. 普通 UDP 服务器编程实现

下面是普通 UDP 服务器典型的服务端和客户端代码的实现。

(1) UDP 服务器服务端代码

udpserver.c 源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
```

```

#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#define MYPOR 4950
#define MAXBUFL 100
int main()
{
    int sockfd;
    struct sockaddr_in my_addr;          /* 存放本地 IP 地址信息*/
    struct sockaddr_in their_addr;       /* 存放连接方 IP 地址信息*/
    int addr_len, numbytes;
    char buf[MAXBUFL];
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket");
        return -1;
    }
    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(MYPOR);
    my_addr.sin_addr.s_addr = INADDR_ANY;
    bzero(&(my_addr.sin_zero), 8);
    if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) \
        == -1) {
        perror("bind");
        return -1;
    }
    addr_len = sizeof(struct sockaddr);
    while(1){
        if ((numbytes=recvfrom(sockfd, buf, MAXBUFL, 0, \
            (struct sockaddr *)&their_addr, &addr_len)) == -1) {
            perror("recvfrom");
            return -1;
        }
        printf("got packet from %s\n",inet_ntoa(their_addr.sin_addr));
        printf("packet is %d bytes length\n",numbytes);
        buf[numbytes] = '\0';
        printf("packet contains \"%s\"\n",buf);
    }
    close(sockfd);
    return 0 ;
}

```

(2) UDP 客户端代码

udptalker.c 源代码如下:

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/socket.h>
#define MYPOR 4950

```

```

int main(int argc, char *argv[])
{
    int sockfd;
    struct sockaddr_in their_addr;
    struct hostent *he;
    int numbytes;
    if (argc != 3) {
        fprintf(stderr, "usage: talker hostname message\n");
        return -1;
    }
    if ((he=gethostbyname(argv[1])) == NULL) { /* 得到主机信息 */
        perror("gethostbyname");
        return -1;
    }
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket");
        return -1;
    }
    their_addr.sin_family = AF_INET;
    their_addr.sin_port = htons(MYPORT);
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    bzero(&(their_addr.sin_zero), 8);
    if ((numbytes=sendto(sockfd, argv[2], strlen(argv[2]), 0, \
        (struct sockaddr *)&their_addr, sizeof(struct sockaddr))) == -1) {
        perror("sendto");
        return -1;
    }
    printf("sent %d bytes to %s\n", numbytes, inet_ntoa(their_addr.sin_addr));
    close(sockfd);
    return 0;
}

```

(3) 编译与执行

① 编译 UDP 服务端代码 `gcc udpserver.c -o udpserver`。

② 编译 UDP 客户端代码 `gcc udptalker.c -o udptalker`。

③ 在一个界面中启动服务端进程 `./udpserver`。

④ 在另一个界面中执行 `./udptalker 127.0.0.1 "ervery day is new day"`，服务端界面提示信息如下：

```

got packet from 127.0.0.1
packet is 21 bytes length
packet contains "ervery day is new day"

```

18.4.2 UDP 广播

1. UDP 广播说明

一个 IP 地址由网络号和主机号组成，主机号为全 1 的 IP 地址是广播地址，应用程序只能通过 UDP 方式发送广播。一般情况下，如果调用 `sendto`，只能向非广播地址发送数据报文。如果要发送广播数据报文，必须告诉内核，可以通过设置 `SO_BROADCAST` 套接口选项来做到这一点，

设置方法如下：

```
int on=1;
setsockopt(sockfd,SOL_SOCKET,SO_BROADCAST,&on,sizeof(int));
```

网卡是链路层的网络设备，能处理物理层和链路层数据。局域网一般都是以太网，以太网具有广播属性，在以太网上传送的帧会被广泛应用到该以太网中所有的节点，即该以太网内所有的网卡都会接收数据，并送到链路层程序进行判断处理。单播和广播的区别在于帧的目的 MAC 地址，广播的 MAC 地址 32 位为全 1（即 FF-FF-FF-FF-FF-FF），单播的 MAC 地址是某联网主机的网卡 MAC 地址（如 00-aa-00-62-c9-09）。单播时网卡驱动判断该目的主机 MAC 地址与自己的 MAC 地址不匹配时，数据链路层程序将不再处理，直接丢弃该数据包；而对于广播地址（即全 1 的地址），链路层将不做 MAC 地址判断，直接往上传送。图 18-13 画出了广播的数据流程图，由图中可以看出，中间这台主机是在 UDP 层（传输层）才将广播数据包丢弃，因为中间这台主机不提供此端口的广播服务。

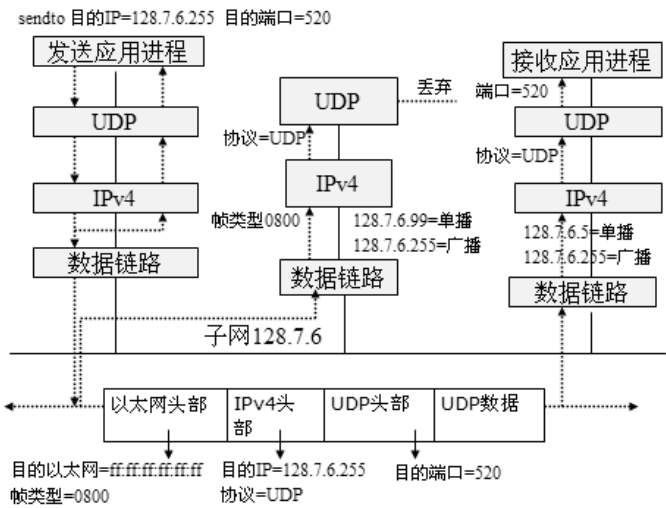


图 18-13 广播物理演示图

2. UDP 广播代码实现

本实例每隔 3 秒钟发送一次广播，将本机时间通知给本子网内所有的主机。

(1) UDP 广播服务端代码

udpbros.c 源代码如下：

```
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define BUFLen 255
int main(int argc,char** argv)
{
    struct sockaddr_in localaddr;
```

```

int sockfd,n;
char msg[BUFLEN+1];
if(argc!=2){
    printf("usage:%s<port>\n",argv[0]);
    return -1;
}
sockfd=socket(AF_INET,SOCK_DGRAM,0);
if(sockfd<0){
    fprintf(stderr,"socket creating error in udpbrocli.c\n");
    return -1;
}
memset(&localaddr,0,sizeof(struct sockaddr_in));
localaddr.sin_port=htons(atoi(argv[1]));
localaddr.sin_addr.s_addr=htonl(INADDR_ANY);
int opt = SO_REUSEADDR;
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
if(bind(sockfd,(struct sockaddr *)&localaddr,sizeof(struct sockaddr_in))<0){
    fprintf(stderr,"bind error in udpbrocli.c\n");
    return -2;
}
n=read(sockfd,msg,BUFLEN);
if(n== -1){
    fprintf(stderr,"read error in udpbrocli.c\n");
    return -3;
}
else{
    msg[n]=0;
    printf("%s\n",msg);
}
}

```

(2) 广播客户端代码

udpbrocli.c 源代码如下:

```

#include <sys/socket.h>
#include <arpa/inet.h>
#include <sys/ioctl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define BUFLEN 255
void getcurtime(char *curtime)
{
    time_t tm;
    time(&tm);
    snprintf(curtime,BUFLEN,"%s\n",ctime(&tm));
}
int main(int argc,char **argv)
{
    struct sockaddr_in peeraddr;
    int sockfd,on=1;
    int num,i;
    char msg[BUFLEN+1];
    if(argc!=3){
        printf("usage:%s<ip address><port>\n",argv[0]);
    }
}

```

```

        return -1;
    }
    sockfd=socket(AF_INET,SOCK_DGRAM,0);
    if(sockfd<0){
        fprintf(stderr,"socket creating error in udpbroser.c\n");
        return -1;
    }
    setsockopt(sockfd,SOL_SOCKET,SO_BROADCAST,&on,sizeof(int));
    memset(&peeraddr,0,sizeof(struct sockaddr_in));
    peeraddr.sin_family=AF_INET;
    if(inet_pton(AF_INET,argv[1],&peeraddr.sin_addr)<=0){
        printf("Wrong dest IP address\n");
        return -1;
    }
    peeraddr.sin_port=htons(atoi(argv[2]));
    for(;;){
        getcurtime(msg);
        num=sendto(sockfd,msg,strlen(msg),0,(struct sockaddr *)&peeraddr,sizeof
(struct sockaddr_in));
        if ( num < 0 )
        {
            perror("sento data error") ;
            return -1 ;
        }
        printf("%s\n",msg);
        sleep(3);
    }
}

```

(3) 编译与执行

- ① 编译 UDP 广播服务端代码 `gcc udpbroser.c -o udpbroser`。
- ② 编译 UDP 广播客户端代码 `gcc udpbrocli.c -o udpbrocli`。
- ③ 在一个界面中执行 `./udpbroser 1234`。
- ④ 在另一个界面中执行 `./udpbrocli 127.0.0.1 1234`。
- ⑤ 在两个界面中执行的结果如下：

Sat Jan 24 06:06:13 2009

18.4.3 UDP 多播

1. UDP 多播说明

单播用于两个主机之间的端对端通信，广播用于一个主机对整个局域网中所有主机上的数据通信。单播和广播是两个极端，要么对一个主机进行通信，要么对整个局域网上的主机进行通信。实际情况下，经常需要对一组特定的主机进行通信，而不是对整个局域网上的所有主机，这就是多播的用途。

多播也称为“组播”，即将网络中同一业务类型的主机进行逻辑上的分组，进行数据收发时

候，其数据仅仅在同一分组中进行，其他的主机没有加入此分组，因此不能收发对应的数据。

广播和多播的 MAC 地址为特殊的 MAC 地址。MAC 地址的特征是网卡驱动判断目的 MAC 地址用途的基础。

IANA（互联网数字分配机构）拥有一个以太网地址块，即高位 24 bit 为 00:00:5e（十六进制数表示），这意味着该地址块所拥有的地址范围从 00:00:5e:00:00:00 到 00:00:5e:ff:ff:ff，IANA 将其中的一半分配为多播地址。为了指明一个多播地址，任何一个以太网地址的首字节必须是 01，这意味着与 IP 多播相对应的以太网地址范围从 01:00:5e:00:00:00 到 01:00:5e:7f:ff:ff。网卡驱动首先判断数据帧目的 MAC 地址是否为广播 MAC 地址（ff:ff:ff:ff:ff:ff），再判断是否为多播 MAC 地址（从 01:00:5e:00:00:00 到 01:00:5e:7f:ff:ff），如果都不是，则是单播地址。单播地址与网卡 MAC 不匹配，数据帧将丢弃。网卡对多播数据在数据链路层并不判断其 MAC 地址是否符合，而是交给 IP 层进行处理。

多播传输中，数据被发送到接收者的多播地址，而不是每个接收者的单播地址，发送者只发送一个数据副本，源端到目标端路径上的中间节点会复制该数据。现在多播 IP 已经广泛应用于网络游戏、视频广播的领域。

多播是通过 D 类地址进行的，D 类地址的前 4 位为 1110，后面 28 位为群播的组标识，地址范围从 224.0.0.0 到 239.255.255.255。下面是特殊的 IPv4 多播地址。

- ① 224.0.0.0 —— 保留。
- ② 224.0.0.1 —— 本子网中的所有的主机。
- ③ 224.0.0.2 —— 本子网中的所有网关。
- ④ 224.0.1.1 —— NTP（网络时间协议）组。
- ⑤ 224.0.0.4 —— 网段中所有的 DVMRP 路由器。
- ⑥ 224.0.0.5 —— 所有的 OSPF 路由器。
- ⑦ 224.0.0.6 —— 所有的 OSPF 指派路由器。
- ⑧ 224.0.0.9 —— 所有的 RIPv2 路由器。

当一个多播分组到达一个以太网时，形成帧后，它的 MAC 地址为 01:00:5e:xx:xx:xx，其后 23 位由多播组标识的后 23 位映射而成。例如，目的地址为 224.0.1.1 的多播分组，在以太网上帧的 MAC 地址就为 01:00:5e:00:01:01，如图 18-14 所示。

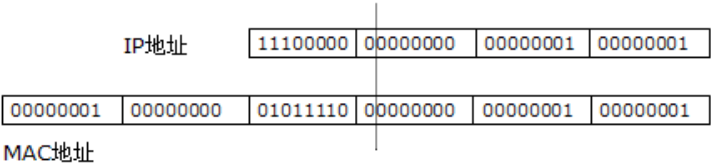


图 18-14 多播 MAC 地址图

由于多播 IP 地址中组标识有 28 位，而映射到 MAC 地址的只有 23 位，还差 5 位，所以有 32 个组将映射成相同的 MAC 地址。例如：224.0.0.1、225.0.1.1、239.128.1.1 都映射到 MAC 地址 01:00:5e:00:01:01。因此，要由 IP 层来检验到达的多播分组是否是自己加入的多播组，如果不是，则抛弃该分组。

图 18-15 画出了多播数据报文在子网中发送和接收的流程图。从图中可以看出，多播数据是在 IP 层进行处理的，IP 层判断主机是否加入此多播组，若没有加入，多播数据包将被丢弃。

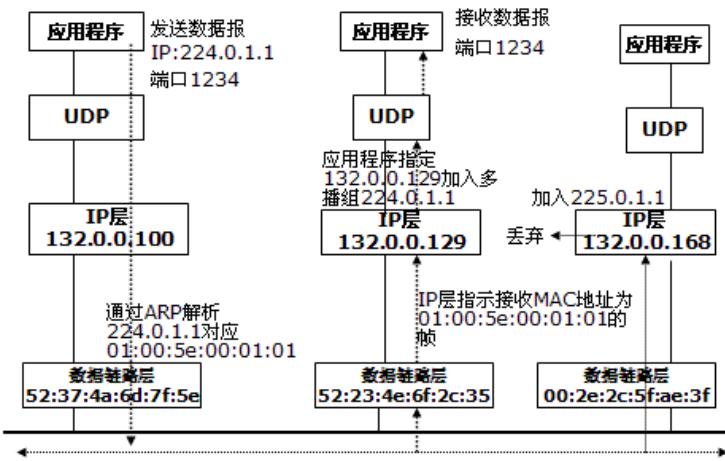


图 18-15 多播原理图

2. UDP 多播代码实现

一个发送端应用程序向多播组地址发送数据报文，一个或多个接收服务端接收该多播组的报文。为了让接收端能接收该多播组的数据报文，接收端应用程序首先要加入这个多播组，加入多播组的方法是设置套接口选项 IP_ADD_MEMBERSHIP，设置方法如下：

```
struct ip_mreq{
    struct in_addr imr_multiaddr; /*IPv4 的 D 类多播地址*/
    struct in_addr imr_interface; /*本地接口 IPv4 地址*/
} mcaddr;
setsockopt(sockfd, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mcaddr, sizeof(struct ip_mreq))
```

下面是使用 Linux 多播技术广播数据的服务器端和客户端代码实现。

(1) 多播服务端代码

udpmulsvr.c 源代码如下：

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
/* 多播 IP 地址 */
```



```

char * host_name = "224.0.0.1";
int port = 6789;
int main(void)
{
    struct ip_mreq command;
    int loop = 1; /* 多播循环 */
    int iter = 0;
    int sin_len;
    char message[256];
    int socket_descriptor;
    struct sockaddr_in sin;
    struct hostent *server_host_name;
    if((server_host_name = gethostbyname(host_name)) == 0)
    {
        perror("gethostbyname");
        return -1 ;
    }
    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = htonl(INADDR_ANY);
    sin.sin_port = htons(port);
    if((socket_descriptor = socket(PF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket");
        return -1 ;
    }
    /* 调用 bind 之前, 设置套接口选项启用多播 IP 支持*/
    loop = 1;
    if(setsockopt(socket_descriptor, SOL_SOCKET, SO_REUSEADDR, &loop, sizeof(loop))
< 0){
        perror("setsockopt:SO_REUSEADDR");
        return -1 ;
    }
    if(bind(socket_descriptor, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
        perror("bind");
        return -1 ;
    }
    /* 设置多播套接口 */
    loop = 1;
    if(setsockopt(socket_descriptor, IPPROTO_IP, IP_MULTICAST_LOOP, &loop,
sizeof(loop)) < 0) {
        perror("setsockopt:IP_MULTICAST_LOOP");
        return -1 ;
    }
    /* 加入一个广播组, 告诉 Linux 内核, 特定的套接口即将接受广播数据*/
    command.imr_multiaddr.s_addr = inet_addr("224.0.0.1");
    command.imr_interface.s_addr = htonl(INADDR_ANY);
    if(command.imr_multiaddr.s_addr == -1) {
        perror("224.0.0.1 not a legal multicast address");
        return -1 ;
    }
    if (setsockopt(socket_descriptor, IPPROTO_IP, IP_ADD_MEMBERSHIP, &command,
sizeof(command)) < 0){
        perror("setsockopt:IP_ADD_MEMBERSHIP");
        return -1 ;
    }
}

```

```

while(iter++ < 3)
{
    sin_len = sizeof(sin);
    if(recvfrom(socket_descriptor, message, 256, 0, (struct sockaddr *)&sin,
&sin_len) == -1) {
        perror("recvfrom");
        return -1 ;
    }
    printf("Response %%-2d from server: %s\n", iter, message);
    sleep(2);
}
if(setsockopt(socket_descriptor, IPPROTO_IP, IP_DROP_MEMBERSHIP, &command,
sizeof(command)) < 0) {
    perror("setsockopt:IP_DROP_MEMBERSHIP");
    return -1 ;
}
close(socket_descriptor);
return 0 ;
}

```

(2) 多播客户端代码

udpmulcli.c 源代码如下:

```

#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
int port = 6789;
int main(void)
{
    int socket_descriptor;
    struct sockaddr_in address;
    /* 首先建立套接口 */
    socket_descriptor = socket(AF_INET, SOCK_DGRAM, 0);
    if (socket_descriptor == -1)
    {
        perror("open socket error");
        return -1;
    }
    /* 初始化 IP 多播地址 */
    memset(&address, 0, sizeof(address));
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = inet_addr("224.0.0.1");
    address.sin_port = htons(port);
    /* 开始进行 IP 多播 */
    while(1) {
        if(sendto(socket_descriptor, "test from multicast", sizeof("test from
multicast"), 0, (struct sockaddr *)&address, sizeof(address)) < 0)
        {
            perror("sendto error");
            return -1;
        }
    }
}

```

```
    }  
    sleep(2);  
}  
return 0 ;  
}
```

(3) 编译与执行

① 编译多播服务端代码 `gcc udpmulsvr.c -o udpmulsvr`。

② 编译多播客户端代码 `gcc udpmulcli.c -o udpmulcli`。

③ 在一个窗口中执行 `./udpmulsvr`。

④ 在另一个窗口中执行 `./udpmulcli`。

⑤ 服务端窗口的执行结果如下：

```
Response #1  from server: test from multicast  
Response #2  from server: test from multicast  
Response #3  from server: test from multicast
```

18.5 原始套接字

18.5.1 原始套接字说明

原始套接字提供了一些使用 TCP 和 UDP 协议不能实现的功能，如使用原始套接字可以读/写 ICMPv4、IGMPv4 分组、ping 程序等，也可以读/写特殊的 IPv4 数据包，内核不处理这些数据报文的 IPv4 协议字段。大部分内核只处理 ICMP、IGMP、TCP、UDP 的数据报，但协议字段还可以为其他值，如 OSPF 直接使用 IP 协议，将 IP 数据报的协议字段设为 89，此时就必须有专门的程序通过原始套接字来处理它们。利用原始套接字还可以创建自定义的 IP 数据报文首部，编写基于 IP 协议的高层网络协议。

1. 原始套接字的创建

利用系统调用 `socket` 创建原始套接字的方法如下：

```
#include <sys/socket.h>  
#include <netinet/in.h>  
int sockfd ;  
const int on = 1;  
sockfd = socket(AF_INET, SOCK_RAW, int protocol);  
setsockopt(sockfd, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on));
```

具体说明如下：

`protocol` 参数一般不为 0，可设为 `IPPROTO_ICMP`、`IPPROTO_TCP`、`IPPROTO_UDP` 等。另外，只有超级用户才能创建原始套接字。

用户可以通过设置 `IP_HDRINCL` 选项来编写自己的 IP 数据报文首部。

可以调用 `bind` 函数绑定原始套接字的本地 IP 地址，此时，所有输出的数据报文将用源 IP 地址（仅当 `IP_HDRINCL` 未设置时）；如果不调用 `bind` 函数，则由内核将源 IP 地址设成外出接口的主 IP 地址。

可以调用 `connect` 函数设置数据报文的目的地地址，此后可直接调用 `write` 或 `send` 函数。

在原始套接字中，`bind`、`connect` 中的端口已经没有任何意义。

2. 原始套接字遵循的规则

（1）原始套接字的输出规则

原始套接字的输出应遵循以下规则：

① 如果套接字已经连接，可以调用 `write`、`writen`、`send` 来发送数据，否则需要调用 `sendto` 或 `sendmsg`。

② 如果 `IP_HDRINCL` 选项未设置，则内核写的数据起始地址指向 IP 头部之后的第一个字节。在这种情况下，内核构造 IP 头部，并将它放在来自进程的数据之前，内核将 IPv4 头部的协议字段设置成用户在调用 `socket` 函数所给的第三个参数。

③ 如果设置了 `IP_HDRINCL`，则内核写的数据起始地址指向 IP 头部的第一个字节，用户所提供的 `data_size` 必须包括头部的字节数。此时应用进程构造除了以下两项外的整个 IP 头部：IPv4 标识字段可以设为 0，要求内核设置该值；IPv4 头部校验和由内核来计算和存储。IPv4 数据报首部各个字段的内容要求均是网络字节序。

（2）内核通过原始套接字接收数据报的规则

内核通过原始套接字接收数据报时，应遵循以下规则：

接收到的 TCP 和 UDP 分组不会传递给原始套接字，如果一个进程希望读取包含 TCP 或 UDP 分组的 IP 数据报，那么它们必须在数据链路层读入。

当内核处理完 ICMP 消息后，绝大部分 ICMP 分组将传递给原始套接字。对源自 Berkeley 的实现，除了回应请求、时间戳请求和地址掩码请求将完全由内核处理以外，所有收到的 ICMP 分组将传递给某个原始套接口。

当内核处理完 IGMP 消息后，所有的 IGMP 分组都将传递给某个原始套接字。

所有带有内核不能识别协议字段的 IP 数据报都将传递给某个原始套接字。

如果数据报以分片形式到达，则该分组将在所有的片段到达并重组后才传给原始套接字。

（3）内核对原始套接字数据报的处理流程

如果在创建原始套接字时，所指定的 `protocol` 参数不为 0，则接收到的数据包的协议字段应与 `protocol` 值匹配，否则该数据报将不传递给该原始套接字。

如果此原始套接字之上绑定了一个本地 IP 地址，那么接收到的数据报的目的 IP 地址应与该绑定地址相匹配，否则该数据报将不传递给该套接字。

如果此原始套接字通过调用 `connect` 指定了一个对方的 IP 地址，那么接收到的数据报的源 IP 地址应与该已连接地址相匹配，否则该数据报将不传递给该套接字。

如果一个原始套接字以 `protocol` 参数为 0 的方式创建，而且未调用 `bind` 或 `connect`，那么对于内核收到的每一个原始数据报，该原始套接字都会收到一份副本。

当接收到的数据报传递给 IPv4 原始套接字时，整个数据报（包括 IP 头部）都将传递给进程；而对于 IPv6，则将去除扩展头部。

18.5.2 原始套接字举例

1. 得到 IP 信息

本实例是利用原始套接字得到网卡的 IP 信息。

(1) 程序代码

`ip.c` 源代码如下：

```
#include <stdio.h>
#include <error.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netdb.h>
#define PACKET_SIZE 4096
#define MAX_WAIT_TIME 5
#define DEST_ADDR "127.0.0.1"
extern int errno;
char sendpacket[PACKET_SIZE];
char recvpacket[PACKET_SIZE];
int sockfd, datalen=56;
struct sockaddr_in dest_addr;
void send_packet();
void recv_packet();
unsigned short cal_chksum(unsigned short * addr, int len);
void showiphdr(struct ip *ip);
void onTerm();
int main(int argc, char *argv[])
{
    struct hostent *host;
    struct protoent *protocol;
```

```

unsigned long inaddr=0L;
if((protocol=getprotobyname("icmp"))==NULL)
{
    perror("unknow protocol icmp");
    return -1;
}
if((sockfd=socket(AF_INET,SOCK_RAW,protocol->p_proto))<0){
    perror("socket error");
    return -2;
}
bzero(&dest_addr,sizeof(dest_addr));
dest_addr.sin_family=AF_INET;
inaddr=inet_addr(DEST_ADDR);
memcpy((char*)&dest_addr.sin_addr,(char*)&inaddr,sizeof(inaddr));
send_packet();
recv_packet();
return 0;
}
void send_packet()
{
    int i,packetsize;
    struct icmp *icmp;
    icmp=(struct icmp*)sendpacket;
    icmp->icmp_type=ICMP_ECHO;
    icmp->icmp_code=0;
    icmp->icmp_cksum=0;
    icmp->icmp_id=getpid();
    icmp->icmp_seq=1;
    packetsize=8+datalen; //数据包的大小
    icmp->icmp_cksum=cal_chksum((unsigned short*)icmp,packetsize);
    if((sendto(sockfd,sendpacket,packetsize,0,(struct sockaddr*)&dest_addr,sizeof
(dest_addr)))<0){
        perror("send ICMP packets error\n");
        exit(3);
    }
    printf("send ICMP packet to %s\n",inet_ntoa(dest_addr.sin_addr));
}
unsigned short cal_chksum(unsigned short *addr,int len)
{
    int nleft=len;
    int sum=0;
    unsigned short *w=addr;
    unsigned short answer=0;
    while(nleft>1){
        sum+=*w++;
        nleft-=2;
    }
    if(nleft==1){
        *(unsigned char*)&answer=*(unsigned char*)w;
        sum+=answer;
    }
    sum=(sum>>16)+(sum&0xffff);
    sum+=(sum>>16);
    answer=~sum;
    return answer;
}

```

```

void recv_packet()
{
    int n, fromlen, packet_no;
    struct sockaddr_in from;
    struct ip *ip;
    struct icmp *icmp;
    signal(SIGALRM, onTerm);
    while(1)
    {
        fromlen = sizeof(from);
        alarm(MAX_WAIT_TIME);
        if((n = recvfrom(sockfd, recvpacket, sizeof(recvpacket), 0, (struct sockaddr*)&from, &fromlen)) < 0) {
            perror("Receive packet error");
            continue;
        }
        ip = (struct ip *)recvpacket;
        showiphdr(ip);
        printf("len:%d", ip->ip_hl);
        icmp = (struct icmp*)(recvpacket + 4 * ip->ip_hl); // 取 ICMP 报头
        printf("ICMP TYPE=%d\n", icmp->icmp_type);
    }
}

void showiphdr(struct ip *ip)
{
    printf("-----IP HEADER-----\n");
    printf("version:%d\n", ip->ip_v);
    printf("header length:%d\n", ip->ip_hl);
    printf("type of service:%d\n", ip->ip_tos);
    printf("total length:%d\n", ip->ip_len);
    printf("identification:%d\n", ip->ip_id);
    printf("fragment offset field:%d\n", ip->ip_off);
    printf("time to live:%d\n", ip->ip_ttl);
    printf("protocol:%d\n", ip->ip_p);
    /*printf("checksum:%s\n", ip->ip_sum );*/
    printf("source IP address:%s\n", inet_ntoa(ip->ip_src));
    printf("destination IP address:%s\n", inet_ntoa(ip->ip_dst));
}

void onTerm()
{
    close(sockfd);
    exit(0);
}

```

(2) 编译与执行

① 编译 `gcc ip.c -o ip`。

② 切换到 root 权限，执行 `./ip`，执行结果如下：

```

send ICMP packet to 127.0.0.1
-----IP HEADER-----
version:4
header length:5
type of service:0
total length:21504

```

```

identification:0
fragment offset field:64
time to live:64
protocol:1
source IP address:127.0.0.1
destination IP address:127.0.0.1

```

2. ping 程序

本实例通过原始套接字实现 ping 命令。

(1) 程序代码

ping.c 源代码如下:

```

#include <stdio.h>
#include <signal.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netdb.h>
#include <setjmp.h>
#include <errno.h>
#include <stdlib.h>
#include <strings.h>
#define PACKET_SIZE    4096
#define MAX_WAIT_TIME  5
#define MAX_NO_PACKETS 3
char sendpacket[PACKET_SIZE];
char recvpacket[PACKET_SIZE];
int sockfd, datalen=56;
int nsend=0, nreceived=0;
struct sockaddr_in dest_addr;
pid_t pid;
struct sockaddr_in from;
struct timeval tvrecv;
void statistics(int signo);
unsigned short cal_chksum(unsigned short *addr, int len);
int pack(int pack_no);
void send_packet(void);
void recv_packet(void);
int unpack(char *buf, int len);
void tv_sub(struct timeval *out, struct timeval *in);
void statistics(int signo)
{
    printf("\n-----PING statistics-----\n");
    printf("%d packets transmitted, %d received , %%d lost\n", nsend, nreceived,
        (nsend-nreceived)/nsend*100);
    close(sockfd);
    exit(1);
}
/*校验和算法*/

```



```

unsigned short cal_chksum(unsigned short *addr,int len)
{
    int nleft=len;
    int sum=0;
    unsigned short *w=addr;
    unsigned short answer=0;

    /*把 ICMP 报头二进制数据以 2B 为单位累加起来*/
    while(nleft>1)
    {
        sum+=*w++;
        nleft-=2;
    }
    /*若 ICMP 报头为奇数个字节, 会剩下最后一字节。把最后一个字节视为一个 2B 数据的高字节, 这个 2B 数据的低字节为 0, 继续累加*/
    if( nleft==1)
    {
        *(unsigned char *)(&answer)=*(unsigned char *)w;
        sum+=answer;
    }
    sum=(sum>>16)+(sum&0xffff);
    sum+=(sum>>16);
    answer=~sum;
    return answer;
}

/*设置 ICMP 报头*/
int pack(int pack_no)
{
    int i,packsize;
    struct icmp *icmp;
    struct timeval *tval;

    icmp=(struct icmp*)sendpacket;
    icmp->icmp_type=ICMP_ECHO;
    icmp->icmp_code=0;
    icmp->icmp_cksum=0;
    icmp->icmp_seq=pack_no;
    icmp->icmp_id=pid;
    packsize=8+datalen;
    tval= (struct timeval *)icmp->icmp_data;
    gettimeofday(tval,NULL); /*记录发送时间*/
    icmp->icmp_cksum=cal_chksum( (unsigned short *)icmp,packsize); /*校验算法*/
    return packsize;
}

/*发送三个 ICMP 报文*/
void send_packet()
{
    int packsize;
    while( nsend<MAX_NO_PACKETS)
    {
        nsend++;
        packsize=pack(nsend); /*设置 ICMP 报头*/
        if( sendto(sockfd,sendpacket,packsize,0,
            (struct sockaddr *)&dest_addr,sizeof(dest_addr) )<0 )
        {
            perror("sendto error");
            continue;
        }
    }
    sleep(1); /*每隔一秒发送一个 ICMP 报文*/
}

/*接收所有的 ICMP 报文*/

```

```

void recv_packet()
{
    int n,fromlen;
    extern int errno;

    signal(SIGALRM,statistics);
    fromlen=sizeof(from);
    while( nreceived<10)
    {
        alarm(MAX_WAIT_TIME);
        if( (n=recvfrom(sockfd,recvpacket,sizeof(recvpacket),0,(struct sockaddr *)
&from,&fromlen)) <0)
        {
            if(errno==EINTR)continue;
            perror("recvfrom error");
            continue;
        }
        gettimeofday(&tvrecv,NULL); /*记录接收时间*/
        if(unpack(recvpacket,n)==-1)continue;
        nreceived++;
    }
}
/*剥去 ICMP 报头*/
int unpack(char *buf,int len)
{
    int i,iphdrln;
    struct ip *ip;
    struct icmp *icmp;
    struct timeval *tvsend;
    double rtt;
    ip=(struct ip *)buf;
    iphdrln=(ip->ip_hl)*4; /*求 ip 报头长度, 即 ip 报头的长度标志乘以 4*/
    icmp=(struct icmp *) (buf+iphdrln); /*越过 ip 报头,指向 ICMP 报头*/
    len-=iphdrln; /*ICMP 报头及 ICMP 数据报的总长度*/
    if( len<8) /*小于 ICMP 报头长度则不合理*/
    {
        printf("ICMP packets\'s length is less than 8\n");
        return -1;
    }
    /*确保所接收的是我所发的 ICMP 的回应*/
    if( (icmp->icmp_type==ICMP_ECHOREPLY) && (icmp->icmp_id==pid) )
    {
        tvsend=(struct timeval *)icmp->icmp_data;
        tv_sub(&tvrecv,tvsend); /*接收和发送的时间差*/
        rtt=tvrecv.tv_sec*1000+tvrecv.tv_usec/1000; /*以毫秒为单位计算 rtt*/
        /*显示相关信息*/
        printf("%d byte from %s: icmp_seq=%u ttl=%d rtt=%.3f ms\n",
            len,
            inet_ntoa(from.sin_addr),
            icmp->icmp_seq,
            ip->ip_ttl,
            rtt);
    }
    else return -1;
}
main(int argc,char *argv[])
{
    struct hostent *host;
    struct protoent *protocol;
    unsigned long inaddr=0l;
    int waittime=MAX_WAIT_TIME;
    int size=50*1024;

```

```

if(argc<2)
{
    printf("usage:%s hostname/IP address\n",argv[0]);
    return -1 ;
}
if( (protocol=getprotobyname("icmp") )==NULL)
{
    perror("getprotobyname");
    return -1 ;
}
/*生成使用 ICMP 的原始套接字, 这种套接字只有 root 才能生成*/
if( (sockfd=socket(AF_INET,SOCK_RAW,protocol->p_proto) )<0)
{
    perror("socket error");
    return -1 ;
}
/* 回收 root 权限, 设置当前用户权限*/
setuid(getuid());
/*扩大套接字接收缓冲区到 50KB, 这样做主要是为了减小接收缓冲区溢出的
可能性, 若无意中 ping 一个广播地址或多播地址, 将会引来大量的应答*/
setsockopt(sockfd,SOL_SOCKET,SO_RCVBUF,&size,sizeof(size) );
bzero(&dest_addr,sizeof(dest_addr));
dest_addr.sin_family=AF_INET;
if((host=gethostbyname(argv[1]))==NULL)
{
    perror("gethostbyname error");
    return -1 ;
}
dest_addr.sin_addr = *((struct in_addr *) host->h_addr);
pid=getpid();
printf("PING %s(%s): %d bytes data in ICMP packets.\n",argv[1],
    inet_ntoa(dest_addr.sin_addr),datalen);
send_packet(); /*发送所有的 ICMP 报文*/
recv_packet(); /*接收所有的 ICMP 报文*/
statistics(SIGALRM); /*进行统计*/
return 0;
}
/*两个 timeval 结构相减*/
void tv_sub(struct timeval *out,struct timeval *in)
{
    if( (out->tv_usec==in->tv_usec)<0)
    {
        --out->tv_sec;
        out->tv_usec+=1000000;
    }
    out->tv_sec-=in->tv_sec;
}

```

(2) 编译和执行

① 编译 `gcc ping.c -o ping`。

② 切换到 root 权限, 执行 `./ping 127.0.0.1`, 执行结果如下:

```

PING 127.0.0.1(127.0.0.1): 56 bytes data in ICMP packets.
64 byte from 127.0.0.1: icmp_seq=1 ttl=64 rtt=1006.000 ms

```

18.6 本地进程间套接字

Linux 本地进程间套接字编程是完全移植和继承 UNIX 本地进程间套接字编程。两者的编程方法和注意事项完全相同，下面只以 UNIX 域套接字为例加以说明。

18.6.1 非命名 UNIX 域套接字管道

1. 非命名 UNIX 域套接字管道说明

非命名 UNIX 套接字域管道适用于父子进程间通信的场合，使用 `socketpair()` 建立这种管道。

`socketpair()` 建立一对已经连接的匿名套接字，其特性由协议簇 `d`、类型 `type`、协议 `protocol` 决定，建立的两个套接字描述符会放在 `sv[0]` 和 `sv[1]` 中。

`socketpair` 函数原型及其说明如下：

socketpair（建立匿名 UNIX 域管道套接字）		
所需头文件	#include <sys/types.h> #include <sys/socket.h>	
函数说明	建立一个 socket 文件描述符	
函数原型	int socketpair(int d, int type, int protocol, int sv[2])	
函数传入值	d	表示协议簇，只能为 AF_LOCAL 或者 AF_UNIX
	type	表示类型，只能为 0
	protocol	SOCK_STREAM: 双向可靠的数据流，用 SOCK_STREAM 建立的套接字对是管道流，与一般的管道相区别的是，套接字对建立的通道是双向的，即每一端都可以进行读写
		SOCK_DGRAM: 双向不可靠的数据报
	sv	建立的两个套接字描述符 sv[0] 和 sv[1]
函数返回值	成功：0	
	失败：-1，失败原因存于 error 中	
错误代码	EAFNOSUPPORT: 指定的地址族本机不支持 EFAULT: 参数 sv 所指不是本进程的合法地址 EMFILE: 本进程使用了过多的描述符 ENFILE: 系统的文件打开总数量已经达到 EOPNOTSUPP: 所指定的协议不能用于建立套接字对 EPROTONOSUPPORT: 所指定的协议本机不支持	

2. 非命名 UNIX 域套接字管道代码举例

（1）程序代码

Socketpair 源代码如下：

```
#include <sys/types.h>
#include <sys/socket.h>
#include <Linux/un.h>
```

```

#include <string.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
void err_sys(const char *errmsg);
int main(void)
{
    int sockfd[2];
    pid_t pid;
    if ((socketpair(AF_LOCAL, SOCK_STREAM, 0, sockfd)) == -1)
        err_sys("Socketpair");
    if ((pid = fork()) == -1)
    {
        err_sys("fork");
    }
    else if (pid == 0)
    {
        /* child process */
        char buf[] = "hello china", s[BUFSIZ];
        ssize_t n;
        close(sockfd[1]);
        write(sockfd[0], buf, sizeof(buf));
        if ((n = read(sockfd[0], s, sizeof(s))) == -1)
            err_sys("read");
        write(STDOUT_FILENO, s, n);
        return 0;
    } else if (pid > 0) { /* parent process */
        char buf[BUFSIZ];
        ssize_t n;
        close(sockfd[0]);
        n = read(sockfd[1], buf, sizeof(buf));
        write(sockfd[1], buf, n);
        return 0;
    }
}
void err_sys(const char *errmsg)
{
    perror(errmsg);
    exit(1);
}

```

(2) 编译和执行

编译 `gcc socketpair.c -o socketpair`。

执行 `./socketpair`，执行结果如下：

```
hello china
```

18.6.2 UNIX 域套接字

1. UNIX 域套接字说明

UNIX 域套接字是在同一台主机上的客户端/服务器通信时使用的一种方法，相对其他方法来

说（例如进程间管道通信），它在形式上与传统套接字 API 的调用方法相同。UNIX 域有两种类型的套接字：字节流套接字和数据报套接字，字节流套接字类似于 TCP，数据报套接字类似于 UDP。UNIX 域套接字的特点为：UNIX 域套接字与 TCP 套接字相比，前者在同一台主机的传输速度是后者的两倍；UNIX 域套接字可以在同一台主机的各进程之间传递描述符，UNIX 域套接字与传统套接字的区别是用路径名来表示协议的描述。

UNIX 域的地址结构在文件 <Linux/un.h> 中，其定义说明如下：

```
#define UNIX_PATH_MAX 108
struct sockaddr_un {
    sa_family_t sun_family;           /*协议名称*/
    char sun_path[UNIX_PATH_MAX];    /*路径名*/
};
```

UNIX 域地址结构成员变量 `sun_family` 的值是 `AF_UNIX` 或者 `AF_LOCAL`，`sun_path` 是一个路径名。结构 `sockaddr_un` 的长度使用宏 `SUN_LEN` 定义，默认大小为 108，`SUN_LEN` 宏的定义如下：

```
# define SUN_LEN(ptr) ((size_t) (((struct sockaddr_un*) 0)->sun_path)+ strlen
((ptr)->sun_path))
```

UNIX 域的套接字函数和以太网套接字（`AF_INET`）的函数相同，但是当用于 UNIX 域套接字时，套接字函数有一些差别和限制，具体如下：

使用函数 `bind()` 进行套接字和地址绑定的时候，地址结构中的路径名和路径名所表示的文件默认访问权限为 0777，即用户、用户所属的组和其他组的用户都能读、写和执行。

结构 `sun_path` 中的路径名必须是一个绝对路径，不能是相对路径。

函数 `connect()` 使用的路径名必须是一个绑定在某个已打开的 UNIX 域套接字上的路径名，而且套接字的类型也必须一致。下列情况将出错：

- 该路径名存在但不是一个套接字。
- 路径名存在且是一个套接口，但没有与该路径名相关联打开的描述字。
- 路径名存在且是一个打开的套接字，但类型不符。

用函数 `connect()` 连接 UNIX 域套接字时的权限检查和用函数 `open()` 以只写方式访问路径名完全相同。

UNIX 域字节流套接字和 TCP 套接字类似，它们都为进程提供一个没有记录边界的字节流接口。

如果 UNIX 域字节流套接字的函数 `connect()` 发现监听套接字的队列已满，会立刻返回一个 `ECONNREFUSED` 错误。这和 TCP 有所不同，TCP 如果监听套接字的队列已满，它将忽略到来的 SYN，TCP 连接的发起方会接着发送几次 SYN 重试。

UNIX 域数据报套接字和 UDP 套接字类似，它们都提供一个保留记录边界的不可靠的数据服务。

2. UNIX 域服务端与客户端代码实现

(1) UNIX 域服务端代码

un_server.c 源代码如下:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
//定义用于通信的文件名
#define UNIX_DOMAIN "/tmp/UNIX.domain"
int main()
{
    socklen_t clt_addr_len;
    int listen_fd;
    int com_fd;
    int ret;
    int i;
    static char recv_buf[1024];
    int len;
    struct sockaddr_un clt_addr;
    struct sockaddr_un srv_addr;
    //创建通信的套接字, 通信域为 UNIX 通信域
    listen_fd=socket(AF_UNIX,SOCK_STREAM,0);
    if(listen_fd<0){
        perror("cannot create listening socket");
        return -1;
    }
    //设置服务器地址参数
    srv_addr.sun_family=AF_UNIX;
    strncpy(srv_addr.sun_path,UNIX_DOMAIN,sizeof(srv_addr.sun_path)-1);
    unlink(UNIX_DOMAIN);
    //绑定套接字与服务器地址信息
    ret=bind(listen_fd,(struct sockaddr*)&srv_addr,sizeof(srv_addr));
    if(ret==-1){
        perror("cannot bind server socket");
        close(listen_fd);
        unlink(UNIX_DOMAIN);
        return -1;
    }
    //对套接字进行监听, 判断是否有连接请求
    ret=listen(listen_fd,1);
    if(ret==-1){
        perror("cannot listen the client connect request");
        close(listen_fd);
        unlink(UNIX_DOMAIN);
        return -1;
    }
    //当有连接请求时, 调用 accept 函数建立服务器与客户端之间的连接
    len=sizeof(clt_addr);
    com_fd=accept(listen_fd,(struct sockaddr*)&clt_addr,&len);
    if(com_fd<0){
        perror("cannot accept client connect request");
        close(listen_fd);
```

```

        unlink(UNIX_DOMAIN);
        return -1;
    }
    //读取并输出客户端发送过来的信息
    printf("\n====info====\n");
    for(i=0;i<3;i++){
        memset(recv_buf,0,1024);
        int num=read(com_fd,recv_buf,sizeof(recv_buf));
        printf("Message from client (%d) :%s\n",num,recv_buf);
    }
    close(com_fd);
    close(listen_fd);
    unlink(UNIX_DOMAIN);
    return 0;
}

```

编译 gcc un_server.c -o un_server。

(2) UNIX 域客户端代码

un_cli.c 源代码如下:

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
//定义用于通信的文件名
#define UNIX_DOMAIN "/tmp/UNIX.domain"
int main(void)
{
    int connect_fd;
    int ret;
    char snd_buf[1024];
    int i;
    static struct sockaddr_un srv_addr;
    //创建通信的套接字, 通信域为 UNIX 通信域
    connect_fd=socket(AF_UNIX,SOCK_STREAM,0);
    if(connect_fd<0){
        perror("cannot create communication socket");
        return -1;
    }
    srv_addr.sun_family=AF_UNIX;
    strcpy(srv_addr.sun_path,UNIX_DOMAIN);
    //连接服务器
    ret=connect(connect_fd,(struct sockaddr*)&srv_addr,sizeof(srv_addr));
    if(ret== -1){
        perror("cannot connect to the server");
        close(connect_fd);
        return -1;
    }
    memset(snd_buf,0,1024);
    strcpy(snd_buf,"message from client");
    //给服务器发送消息
    for(i=0;i<3;i++){
        write(connect_fd,snd_buf,sizeof(snd_buf));
    }
}

```



```
    close(connect_fd);  
    return 0;  
}
```

(3) 编译与执行

- ① 编译服务端代码 `gcc un_server.c -o un_server`。
- ② 编译客户端代码 `gcc un_cli.c -o un_cli`。
- ③ 在一个用户界面启动 UNIX 域套接字服务端进程: `./un_server`。
- ④ 在另一个用户界面启动 UNIX 域套接字客户端进程: `./un_cli`。
- ⑤ 在服务端界面显示结果如下:

```
====info====  
Message from client (1024)) :message from client  
Message from client (1024)) :message from client  
Message from client (1024)) :message from client
```

18.7 I/O 编程模型

在 UNIX/Linux 下, 有下面五种 I/O 操作方式。

① 阻塞 I/O 模型: 在这种模型下, 若所调用的 I/O 函数没有完成相关的功能, 就会使进程挂起, 直到有相关数据到达才会返回。如对管道设备、终端设备和网络设备进行读写时经常会出现这种情况。

② 非阻塞模型: 在这种模型下, 当请求的 I/O 操作不能完成时, 不让进程睡眠, 而且返回一个错误。

③ I/O 复用模型: 在这种模型下, 如果请求的 I/O 操作阻塞, 且它不是真正阻塞 I/O, 而是让其中的一个函数等待, 在这期间, I/O 还能进行其他操作。如本节要介绍的 `select` 函数就属于这种模型。

④ 信号驱动 I/O 模型: 在这种模型下, 通过安装一个信号处理程序, 系统可以自动捕获特定信号的到来, 从而启动 I/O, 这是由内核通知用户何时可以启动一个 I/O 操作决定的。这种 I/O 模型在应用编程中很少见, 在此不再介绍。

⑤ 异步 I/O 模型: 在这种模型下, 当一个描述符已准备好, 可以启动 I/O 时, 进程会通知内核。这种 I/O 模型在应用编程中很少见, 在此不再介绍。

1. 阻塞 I/O 模型说明

阻塞 I/O 模式是使用最普遍的 I/O 模式, 大部分程序使用的都是阻塞 I/O 模式。默认的一个套接字建立后, 所处的模式就是阻塞 I/O 模式。

读操作中的 `read`、`readv`、`recv`、`recvfrom`、`recvmsg` 是阻塞函数, 写操作中的 `write`、

`writew`、`send`、`sendmag` 是阻塞函数，另外，`accept`、`connect` 也是阻塞函数。

一般来说，程序进行读阻塞操作有以下两步。

- ① 等待有数据可以读。
- ② 将数据从系统内核中复制到程序的数据区。

对于一个对套接字的输入操作来说，第①步是等待数据从网络上传到本地，当数据包到达的时候，数据将会从网络层复制到内核的缓存中。第②步是从内核中把数据复制到程序的数据区中。

（1）读阻塞流程说明

① 进程调用 `read` 函数从套接字上读取数据，当套接字的接收缓冲区中还没有数据可读时，函数 `read` 将发生阻塞。它会一直阻塞下去，直到套接字的接收缓冲区中有数据可读。

② 经过一段时间后，缓冲区内接收到数据，于是内核便去唤醒该进程，通过 `read` 访问这些数据。

③ 在进程阻塞的过程中，如果对方发生故障，那么这个进程将永远阻塞下去。

写阻塞与读阻塞类似，故在此不再说明。

（2）阻塞 I/O 超时控制

阻塞式 I/O 是编程中最常用的模式，但一般需要进行超时处理。超时处理有三种实现方法：一是用 `alarm` 函数发送信号来实现，二是用 `select` 函数来实现，三是用 `setsockopt` 函数来设置超时。

2. 非阻塞 I/O 模型说明

当把一个套接字设置为非阻塞模式的时候，就相当于告诉了系统内核：“如果请求的 I/O 操作不能够马上完成，进程不进行休眠，请马上返回一个错误给应用程序”。当一个应用程序使用了非阻塞模式的套接字，它需要使用一个循环来不停地测试一个文件描述符是否有数据可读（称为 `polling`）。应用程序不停地 `polling` 内核来检查 I/O 操作是否已经就绪，这将是一个极浪费 CPU 资源的操作。这种模式在实际应用中不是很普遍。

非阻塞模式的实现可以利用 `fcntl` 函数完成。当开始建立一个套接字描述符的时候，系统内核默认设置为阻塞状态。如果不想某个文件描述符处于阻塞状态，可以使用函数 `fcntl` 设置一个套接字标志 `O_NONBLOCK` 来实现非阻塞。实现方法如下：

```
int flag;
flag = fcntl(sockfd, F_GETFL, 0);
flag |= O_NONBLOCK;
fcntl(sockfd, F_SETFL, flag);
```

3. 多路复用 I/O 模型

（1）多路复用 I/O 模型说明

如果一个或多个 I/O 条件满足时，就被通知到，这个能力被称为 I/O 复用，是由函数 `select`

支持的。

I/O 多路复用典型的应用场合有：当客户处理多个文件描述符时（一般是交互式输入和网络套接口），必须使用 I/O 复用。当一个客户同时处理多个套接字时，这种情况是可能的，但很少出现。如果一个 TCP 服务器既要处理监听套接字，又要处理已连接的套接口，一般要用到 I/O 复用。如果一个服务器既要处理 TCP，又要处理 UDP，一般使用 I/O 复用。如果一个服务器要处理多个服务或多个协议，一般要使用 I/O 复用。

在 I/O 多路复用的使用场合中，应用程序需同时处理多路输入/输出流，若采用阻塞模式，将达不到预期目的。若采用非阻塞模式，对多个输入进行轮询，又太浪费 CPU 时间。若设置多个进程，分别处理一条数据通路，将新产生进程间同步与通信问题，使程序变得更加复杂，比较好的方法是使用 I/O 多路复用。

在 I/O 多路复用时，需先构造一张有关描述符的表，然后调用 select 函数，它要等到这些描述符中已有一个准备好的 I/O 才返回，返回时，它告诉进程是哪个文件描述符的 I/O 已经准备好。

当使用 I/O 多路技术的时候，调用 select 函数会阻塞，但不会像 I/O 阻塞模型中调用 send、recv 等函数时的阻塞。当 select 函数返回的时候，也就是有文件描述符可以读写数据的时候，此时可以用读写函数完成数据读写。

和阻塞模式相比，多路复用的高级之处在于，它能同时等待多个文件描述符，当这些文件描述符中的任意一个进入就绪状态时，select 函数就可以返回。

(2) select 函数说明

select 函数原型如下：

select (I/O 多路复用)		
所需头文件	#include <sys/time.h> #include <sys/types.h> #include <unistd.h>	
函数说明	select()用来等待文件描述符状态的改变。参数 n 代表最大的文件描述符加 1，参数 readfds、writefds 和 exceptfds 称为描述符组，用来回传该描述词的读、写或例外的状况	
函数原型	int select(int numfds,fd_set*readfds,fd_set*writefds,fd_set*exceptfds,struct timeval*timeout)	
函数传入值	numfds: 需要检查的号码最高的文件描述符加 1	
	readfds: 由 select()监视的读文件描述符集合	
	writefds: 由 select()监视的写文件描述符集合	
	exeptfds: 由 select()监视的异常处理文件描述符集合	
	timeout	NULL: 永远等待。直到捕捉到信号或文件描述符已准备好为止。如果捕捉到一个信号，则 select 返回-1， errno 设置为 EINTR timeout->tv_sec==0 && timeout->tv_usec==0 完全不等待。测试所有指定的描述符并立即返回。这是得到多个描述符的状态而不阻塞 select 函数的轮询方法 timeout->tv_sec!=0 && timeout->tv_usec!=0 等待指定的秒数和微秒数。当指定的描述符之一已准备好，或当指定的时间值已经超过时立即返回。如果在超时时尚还没有一个描述符准备好，则返回值是 0

续表

select (I/O 多路复用)	
函数返回值	成功: 准备好的文件描述符
	失败: -1, 失败原因的代码存放在 error 中
错误代码	EBADF: 文件描述符为无效的或该文件已关闭 EINTR: 此调用被信号中断 EINVAL: 参数 n 为负值 ENOMEM: 核心内存不足
常见程序片段	<pre>fs_set readset; FD_ZERO(&readset); FD_SET(fd,&readset); select(fd+1,&readset,NULL,NULL,NULL); if(FD_ISSET(fd,readset)) {...}</pre>

select 函数中的 timeout 是一个 struct timeval 类型的指针, 这个时间结构体的精确度可以设置到微秒级。该结构体如下:

```
struct timeval {  
    long tv_sec; /* second */  
    long tv_unsec; /* and microseconds*/  
}
```

在使用 select 函数的过程中需要对文件描述符进行分类处理, 对文件描述符的处理主要涉及四个宏函数, 表 18-4 列出了这四个宏函数及其作用说明。

表 18-4 select 文件描述符集处理宏

宏	说 明
FD_ZERO(fd_set *set)	清除一个文件描述符集
FD_SET(int fd,fd_set *set)	将一个文件描述符加入文件描述符集中
FD_CLR(int fd,fd_set *set)	将一个文件描述符从文件描述符集中清除
FD_ISSET(int fd,fd_set *set)	测试该集中的一个文件描述符有无发生变化

一般来说, 在使用 select 函数之前, 首先使用 FD_ZERO 和 FD_SET 来初始化文件描述符集, 在使用了 select 函数时, 可循环使用 FD_ISSET 测试哪一个文件描述符就绪, 在执行完相关的文件描述符后, 可使用 FD_CLR 来清除该文件描述符。

(3) select 函数举例

本实例用 select 函数完成对两个文件描述符的多路访问, 其中一个为读文件描述符, 一个为写文件描述符。读文件描述符加入到描述符集 inset1, 写描述符集加入到描述符集 inset2, 然后使用 select 函数完成 I/O 多路复用的处理。在 socket 编程中, 使用方法与本程序类似, 只是需要把这这里的文件描述符改成 socket 文件描述符。

select.c 源代码如下:

```

#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>
int main(void)
{
    int fds[2];
    char buf[7+1];
    int i,rc,maxfd;
    fd_set inset1,inset2;
    struct timeval tv;
    if((fds[0] = open ("hello1", O_RDWR|O_CREAT,0666))<0)
    {
        perror("open hello1");
        return -1 ;
    }
    if((fds[1] = open ("hello2", O_RDWR|O_CREAT,0666))<0)
    {
        perror("open hello2");
        return -1 ;
    }
    if((rc = write(fds[0],"Hello!\n",7)))
    {
        printf("rc=%d\n",rc);
    }
    /*取出两个文件描述符中的较大者*/
    maxfd = fds[0]>fds[1] ? fds[0] : fds[1];
    /*初始化读集合 inset1, 并在读集合中加入相应的描述集*/
    FD_ZERO(&inset1);
    FD_SET(fds[0],&inset1);
    /*初始化写集合 inset2, 并在写集合中加入相应的描述集*/
    FD_ZERO(&inset2);
    FD_SET(fds[1],&inset2);
    tv.tv_sec=9; /*设置 select 阻塞等待最长的秒数*/
    tv.tv_usec=0;
    i=0 ;
    while(1){
        /*select 有准备就绪的文件描述符就返回, 否则等待 9s 后报错*/
        if(select(maxfd+1,&inset1,&inset2,NULL,&tv)<0)
        {
            perror("select");
            return -1 ;
        }
        else{
            if(FD_ISSET(fds[0],&inset1)){
                lseek(fds[0],0,SEEK_SET); /*由于读指针已到文件尾, 把读指针重新指向文件头*/
                rc = read(fds[0],buf,7);
                if(rc>0){
                    buf[rc]='\0';
                    printf("read: %s\n",buf);
                }else
                {
                    perror("read error");
                    return -1 ;
                }
            }
        }
    }
}

```

```
    }  
  }  
  if(FD_ISSET(fds[1],&inset2)){  
    i++ ;  
    sprintf(buf,"%3.3s%03d\n","num",i) ;  
    rc = write(fds[1],buf,7);  
    if(rc>0){  
      buf[rc]='\0';  
      printf("rc=%d,write: %s\n",rc,buf);  
    }else  
    {  
      perror("write error");  
    }  
    sleep(10);  
  }  
}  
}  
return 0;  
}
```

编译 gcc select.c -o select。

执行./select，执行结果如下：

```
rc=7  
read: Hello!  
  
rc=7,write: num001  
.....
```

第 6 篇

XML 编程

❖ 第 19 章 XML 概念与语法

❖ 第 20 章 libxml 编程

学海聆听：

- 量力而行，尽力而为；欲速则不达，顺其自然。
- 投资未来。
- 腾蛇无足而飞，鼯鼠五技而穷；先专才，后通才。
- 观念的改变导致行为的改变，行为的改变导致结果的改变。
- 梅须逊雪三分白，雪却输梅一段香；认识你自己。
- 知人者智，自知则明；尺有所短，寸有所长；发挥自己的特长，扬长避短。
- 学会选择，学会珍惜，学会放弃；有时选择比努力更重要。
- 闻道有先后，术业有专攻。
- 人的一生虽然漫长，但关键处只有几步，走好那几步会影响你一生。
- 人不能两次踏入同一条河流，太阳每天都是新的。
- 天下事有难易乎，为之，则难者亦易矣；不为，则易者亦难矣。
- 认可自己，提升自己，完善自己，成就自己。

第 19 章

XML 概念与语法

本章分三个部分，包括 XML 概述、XML 语法和 XPath 语法。XML 概述章节将介绍什么是 XML、XML 与 HTML 的主要区别以及 XML 组成等。XML 语法章节介绍 XML 文档、属性、元素、注释、转义、CDATA、Namespace、entity、DTD、XML 编码的语法格式。XPath 章节介绍有关 XPath 表达式的相关语法。

19.1 XML 概述

1. XML 的概念

关于 XML 的定义，有以下几种说法：

- ① XML 是可扩展标记语言（Extensible Markup Language）的缩写。
- ② XML 是一种类似于 HTML 的标记语言。
- ③ XML 是描述数据的，重点描述“数据是什么”。
- ④ XML 的标记不是在 XML 中预定义的，用户必须定义自己的标记。
- ⑤ XML 使用文档类型定义（DTD）或者模式（Schema）来描述数据。
- ⑥ XML 使用 DTD 或者 Schema 后就是自描述语言。
- ⑦ XML 数据和格式是分离的，XML 文档本身不定义如何来显示数据。
- ⑧ XML 并不只是标记语言，它还可以用来创建新的标记语言（比如 HTML）。
- ⑨ 不能用 XML 来直接编写网页。即便是包含了 XML 数据，依然要转换成 HTML 格式，才能在浏览器上显示。
- ⑩ XML 可以交换数据、共享数据、存储数据、利用数据、创建新的语言。

2. XML 和 HTML 的主要区别

XML 和 HTML 的主要区别如下：

- ① XML 不是 HTML 的替代品，XML 和 HTML 是两种不同用途的语言。
- ② XML 是被设计用来描述数据的。重点是：什么是数据，如何存放数据。
- ③ HTML 是被设计用来显示数据的。重点是：显示数据以及如何更好地显示数据。
- ④ HTML 是与显示信息相关的，XML 则是与描述信息相关的。
- ⑤ HTML 将数据和显示混在一起，而 XML 则将数据和显示分开来。
- ⑥ XML 是元语言，HTML 是具体的语言。XML 可以定义一种新的语言，而 HTML 则不能。人们可以用 XML 语法定义出新的语言，如 XML 是 WAP 和 WML 语言的母亲。

3. XML 与 HTML 举例说明

XML 可以用来描述数据，重点是“数据是什么”。HTML 则是用来显示数据，重点是“如何显示数据”。

HTML 是一个定型的标记语言，它用固有的标记来描述、显示网页内容。比如，<H1>表示首行标题，有固定的尺寸。相对的，XML 则没有固定的标记，不能描述网页具体的外观和内容，它只是描述内容的数据形式和结构。

在 HTML 中，有许多固定的标记，在使用时，不能使用 HTML 规范里没有的标记。而在 XML 中，用户能建立任何自己需要的标记。比如，你的文档里包含一些游戏的攻略，你可以建立一个名为<game>的标记，然后在<game>下根据游戏类别建立<RPG>、<SLG>等标记。只要这些标记清晰，易于理解，你可以建立任何数量的标记。

下面以 test.html、test.xml 来说明两者的不同。

test.html

```
<html>
  <head>
    <TITLE>A Simple HTML Example</TITLE>
  </head>
  <body>
    <H1>HTML is Easy To Learn</H1>
    <P>Welcome to the world of HTML,This is the first paragraph.</P>
  </body>
</html>
```

test.xml

```
<note>
  <to>Lin</to>
  <from>Ordin</from>
  <heading>Reminder</heading>
```

```
<body>Don't forget me this weekend!</body>
</note>
```

对 test.html 和 test.xml 的说明如下:

test.html 中的 html、head、body、h1、p 这些标的类型、含义及用法已经是明确的, 且也是固定的。HTML 每一版本标记的类型、用法及含义由 W3C 进行公布, 全世界遵守这个统一标准。

test.xml 中的 note、to、from、heading、body 这些标记的类型、含义及用法可以自行定义, 通信的多方自行约定含义和约定解析。

如 IE、firefox 浏览器能根据 HTML 标记 (如 title、body) 解析和显示 HTML 文件。而在只有一个 XML 内容文件下, 浏览器无法解析 XML 标记和内容。因为 HTML 标记语法语义已经确定, 而 XML 标记要自己定义语法和语义。

4. XML 的使用

XML 的使用场合如下:

- ① XML 是被设计用来存储数据、携带数据和交换数据的。
- ② XML 可以从 HTML 中分离数据。通过 XML, 可以在 HTML 文件之外存储数据。
- ③ XML 用于交换数据。通过 XML, 可以在不兼容的系统之间交换数据。
- ④ 使用 XML, 可以在网络中交换电子商务信息。
- ⑤ XML 可以用于共享数据。通过 XML, 纯文本文件可以用来共享数据。
- ⑥ XML 可以充分利用数据。使用 XML, 你的数据可以被更多的用户使用。
- ⑦ XML 可以用于创建新的语言, 它是 WAP 和 WML 语言的母亲。
- ⑧ 如果开发者有足够的预见性, 那么将来的应用程序都应该使用 XML 的形式来存储数据。

5. XML 的优势

HTML 是用来设计人机交流功能的, 对布局、外观方面很擅长, 但缺乏对内容 (也就是信息含义) 的表达。除了少数几个用来表达内容或文义的标签, 如 <p>、<address>、<title>、 外, 几乎全都是用来设计网页格式的。假设需要一个能将商品价格明确标示的 <price> 标签, HTML 则无能为力。

XML 和 HTML 的不同处在于: 在 XML 文件中, 我们可以自由定义标签, 并且定义出来的标签可以按自己的意思充分表达文件的内容, 譬如可以定义 <name>、<bookinfo> 这样意义明确的标签。在 XML 中, 只注重内容, 这和 HTML 强调布局的做法不大相同。XML 文件的内容和外观设计是完全分开的, 外观变动时, XML 文件完全不受影响。

同时, XML 的自描述性有利于资料的交换和传递, 商务往来的公司之间, 用不着、也不需要知道对方内部采用何种格式储存资料, 大家都用 XML 作为中介格式即可。这样, 某个系统内部的

变更，并不会影响和它交流往来的其他系统，因而，XML 提供了一层理想的缓冲。很多人认为，XML 将是电子商务理想的格式标准。

XML 除了上述描述数据的优势外，还具有如下优势：XML 可以广泛地运用于 Web 的任何地方，XML 可以满足网络应用的需求，使用 XML 将使编程更加简单，XML 便于学习和创建，XML 文件代码更清晰和便于阅读理解。

6. XML 组成

XML 由 XML 对象组成。XML 对象可以是元素、属性、CDATA、附加内容、文本节点、注释和处理指令。XML 对象又可分为包含“简单内容”和包含“复杂内容”两类，有子节点的 XML 对象归入包含复杂内容的一类。如果 XML 对象是属性、注释、处理指令或文本节点中的任何一个，我们就说它包含简单内容。

7. XSL

XSL是指可扩展样式表语言（EXtensible Stylesheet Language）。它用于将XML文档的内容转换成另一种形式的文档，转换后的文档能被一些应用软件更好地显示。XSL和XML的关系就像CSS和HTML的关系。XML用于承载数据，而XSL则用于设置数据格式。XSL专门用于处理XML文档，并且遵循XML语法。因此，它只能在支持XML的应用程序中与XML结合使用。图 19-1 画出了XSL与其他语言的层次关系。

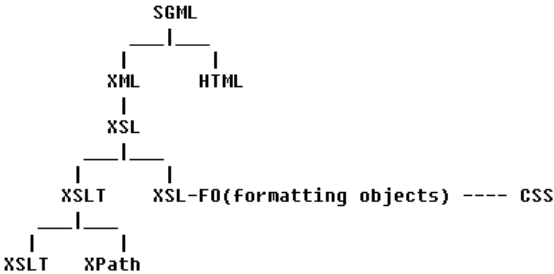


图 19-1 XSL 树形层次图

XSL 包括以下三部分。

- ① XSLT: XSLT 是一种转换 XML 文档的语言，如将 XML 文档转换成 HTML 文档或 PDF 文档等。
- ② XPath: 是在 XML 文档中查找定位信息的语言。
- ③ XSL-FO: 一种用于格式化 XML 文档的语言。

19.2 XML 语法

1. XML 文档的组成

XML 文档包含以下三部分：

- ① 一个 XML 文档声明。

② 一个关于文档类型的定义。

③ 用 XML 标记创建的内容。

下面以 myfile.xml 为例进行说明。

```
<?xml version="1.0" encoding="GB2312"?>
<!DOCTYPE myfile SYSTEM "myfile.dtd">
<myfile>
  <title>XML 轻松学习手册</title>
  <author>ajie</author>
  <email>ajie@aolhoo.com</email>
  <date>20010115</date>
</myfile>
```

其中，第一行<?xml version="1.0"?>就是一个 XML 文档的声明，第二行说明这个文档是用 myfile.dtd 来定义文档类型的，第三行以下的语句就是内容主体部分。

2. XML 文档术语

① 标记 (Tag): 用来定义元素名称和属性名称。在 XML 中，标记必须成对出现，将数据包围在中间。

② 元素 (Element): 由元素名称和元素内容组成。一个元素由一个标记来定义，包括开始和结束标识以及其中的内容。

③ 属性 (Attribute): 由属性名称和属性内容组成。属性是对元素进一步的描述和说明，一个元素可以有多个属性。属性是一种为元素加入描述性信息的机制。

④ 元素的包含关系: 元素 (属性, 子元素 (属性, 子元素))。

⑤ 空元素: 没有内容的元素。书写格式为<Element></Element>或<Element/>。

⑥ 声明 (Declaration): 在所有 XML 文档的第一行都有一个 XML 声明。这个声明表示这个文档是一个 XML 文档，它表示遵循的是哪一个 XML 规范版本。

⑦ DTD (文件类型定义): DTD 用来定义 XML 文档中元素、属性以及元素之间的关系。通过 DTD 文件可以检测 XML 文档结构是否正确，但建立 XML 文档并不一定需要 DTD 文件。

⑧ Well-formed XML (良好格式的 XML): 一个遵守 XML 语法规则，并遵守 XML 规范的文档称为“良好格式”。

⑨ Valid XML (有效格式的 XML): 一个遵守 XML 语法规则，并遵守相应的 DTD 文件规范的 XML 文档称为有效的 XML 文档。

⑩ XML 文档举例说明。

```
<?xml version="1.0" standalone="yes" encoding="UTF-8"?>
<author sex="female">ajie</author>
```

上述 XML 文件中，元素标记 (即元素名称) 为 author，属性标记 (即属性名称) 为 sex。

元素 `author` 内容为 `ajie`，该元素 `author` 的 `sex` 属性内容为 `female`。第一行是 XML 声明。

3. 属性与子元素的比较

属性也可以改成子元素，改成子元素后易于扩充和编程，但减少了可读性。

下面两个文件中，第一个包含 `sex` 属性，第二个把属性 `sex` 改成了子元素。

```
<person sex="female">
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

```
<person>
  <sex>female</sex>
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

属性的缺点有：属性不能包含多个值，子元素却可以包含多个值；属性不容易扩展，并且不能够描述结构，而子元素则可以；属性值很难通过 DTD 进行测试。

属性的优点是增加了文件的可读性，程序处理耗费的内存空间和时间比子元素少。

数据既可以存储在子元素中，也可以存储在属性中。元数据（与数据有关的数据）应该以属性的方式存储，而数据本身应该以元素的形式存储。

4. XML 文档的语法规则

在 XML 文档中，必须有 XML 声明语句。声明语句是文件的第一句，格式为 `<?xml version="1.0" standalone="yes/no" encoding="UTF-8"?>`。声明的作用是告诉浏览器或者其他处理程序这个文档是 XML 文档。声明语句中的 `version` 表示文档遵守 XML 规范的版本，`standalone` 表示文档是否附带 DTD 文件，`encoding` 表示文档所用的语言编码，默认是 UTF-8。

在 XML 文档中，所有的标记必须要有结束标记，包括空标记。

所有的 XML 文档必须有一个根元素，XML 文档中的第一个元素就是根元素。

所有的 XML 文档都必须包含一个单独的标记来定义，所有的其他元素都必须成对地在根元素中嵌套，XML 文档有且只有一个根元素。所有的 XML 标记都必须嵌套合理。

属性值必须使用引号（包括双引号和单引号）。在 XML 中，元素的属性值没有引号引起来是不符合规定的。

使用 XML 解析器解析 XML 文档时，空白部分不会被解析器自动删除。

使用 XML 解析器解析 XML 时，CR/LF 会被转换为 LF（Line Feed，即换行）。

在 XML 中注释的语法基本上和 HTML 中的一样，如 `<!-- 这是一个注释 -->`。

XML 标记都是大小写敏感的。

XML 文档使用了自描述和简单的语法。

5. XML 元素的语法

XML 元素的语法规则如下：

- ① XML 元素是可以扩展的，它们之间有关联。
- ② XML 元素有简单的命名规则。
- ③ XML 元素是可以扩展的，XML 元素的可扩展决定了 XML 文档的可扩展。
- ④ XML 元素是相互关联的，XML 元素之间可以是父子关系和兄弟关系。
- ⑤ XML 元素由元素名称和元素内容组成。XML 元素可以有不同的内容。
- ⑥ 元素可以用来标记 XML 文件中的区段。XML 元素拥有下列格式：

```
<ElementName>Content</ElementName>
```

其中，Content 部分将会被包含在 XML 标记中。

⑦ 虽然 XML 标记通常包围住内容部分，但是也可以建立不含内容的元素，称为空（empty elements）元素。在 XML 中，空元素可以利用下面两种方式来呈现：

```
<ElementName/>  
<ElementName></ElementName>
```

6. XML 元素命名规则

XML 元素的命名规则如下：

- ① 元素的名字可以包含字母、数字和其他字符。
- ② 元素的名字只能以字母和下划线“_”开头。
- ③ 元素的名字不能以 XML（或者 xml、Xml、xMl 等）开头。
- ④ 元素的名字不能包含空格，中间不能包含“:”（冒号）。

7. 属性的语法

XML 属性的语法规则如下：

- ① XML 元素在开始标记处可以有元素属性。
- ② XML 元素可以有属性，属性通常包含一些关于元素的额外信息。
- ③ 属性值必须用引号引起来（单引号、双引号都可以使用，单引号中可以包含双引号）。
- ④ 通常可以用子元素代替属性。

⑤ XML 属性定义格式如下：

```
<person sex="female">why</person>
```

代码中的 person 是元素名称，sex 是属性名称。

8. 注释的语法

XML 注释的语法格式如下：

```
<!-- 这里是注释信息 -->
```

9. 转义字符

不合法的 XML 字符必须替换为相应的实体，然后才可以使用。

如果在 XML 文档中使用类似“<”的字符，那么解析器将会出现错误，因为解析器会认为这是一个新元素的开始。所以，不应该像下面这样书写代码：

```
<message>if salary < 1000 then</message>
```

为了避免出现这种情况，必须将字符“<”转换成实体，改写方法如下：

```
<message>if salary &lt; 1000 then</message>
```

表 19-1 是在 XML 文档中预定义好的五个实体。实体必须以符号“&”开头，以符号“;”结尾。注意：只有“<”字符和“&”字符对于 XML 来说是严格禁止使用的，剩下的都是合法的。为了减少出错，使用实体是一个好习惯。

表 19-1 XML 预定义实体表

实 体	符 号	含 义
<	<	小于号
>	>	大于号
&	&	和
'	'	单引号
"	"	双引号

10. CDATA 的语法

CDATA 的全称是 character data，即字符数据。在标记 CDATA 下，所有的标记、实体引用都被忽略，而被 XML 处理程序统一当做字符数据看待。CDATA 的语法格式如下：

```
<![CDATA[这里放置需要显示的字符]]>
```

例如，<![CDATA[<AUTHOR sex="female">ajie</AUTHOR>]]>在页面上显示的内容为“<AUTHOR sex="female">ajie</AUTHOR>”。

对 CDATA 的两点说明如下：

在XML文档中的所有文本都会被解析器解析，只有在CDATA部件内的文本会被解析器忽略。

CDATA 部件之间不能再包含 CDATA 部件（不能嵌套）。

11. Namespaces 的语法

Namespaces 即为命名空间。命名空间有什么作用呢？当在一个 XML 文档中使用多个 DTD 文件时，就会出现这样的矛盾：因为 XML 中的标识都是自己创建的，在不同的 DTD 文件中，标识名可能相同，但表示的含义不同，这就可能引起数据混乱。

命名空间说明标识属于哪一空间，避免相同命名引起的混乱问题。

Namespaces 通过给标识名称加一个网址（URL）定位的方法来区分这些名称相同的标识。Namespaces 同样需要在 XML 文档的开头部分进行声明。

下列示例的命名空间是 h，解析时用 `http://www.w3.org/TR/html4/` 网址的 DTD 文件解析。

```
<h:table xmlns:h="http://www.w3.org/TR/html4/">
<h:tr>
<h:td>Apples</h:td>
<h:td>Bananas</h:td>
</h:tr>
</h:table>
```

12. entity 的语法

entity 即为“实体”。实体把文档中的共同部分抽象出来，达到一次定义多次引用的目的。用户可以预先定义一个 entity，然后在一个文档中多次调用。

使用 entity 的好处有如下两点：

- ① 它可以减少差错，文档中多个相同的部分只需要输入一次就可以。
- ② 它提高了维护效率。

XML 定义了两类 entity。一种是普通 entity，在 XML 文档中使用；另一种是参数 entity，在 DTD 文件中使用。

entity 的定义语法为：

```
<!DOCTYPE filename [
  <!ENTITY entity-name "entity-content"
]>
```

定义好的 entity 在文档中的引用语法为：

`&entity-name;`

下文 `copyright.xml` 中定义了 `copyright` 实体并加以引用，其文件内容如下：

```
<?xml version="1.0" encoding="GB2312"?>
<!DOCTYPE copyright [
  <!ENTITY copyright "Copyright 2001, Ajie. All rights reserved">
]>
```



```
<myfile>
  <title>XML</title>
  <author>ajie</author>
  <email>ajie@aolhoo.com</email>
  <date>20010115</date>
  &copyright;
</myfile>
```

13. XML 的结构化

XML 促使文档结构化，即所有的信息按某种关系排列。结构化就是为文档建立树形框架，使每一部分都紧密联系，形成一个整体。结构化有如下两个原则：

- ① 每一部分（每一个元素）都和其他元素有关联，关联的级数就形成了结构。
- ② 标识本身的含义与它描述的信息相分离。

14. XML 的确认

符合语法的 XML 文档称为结构良好的 XML 文档，一个结构良好的 XML 文档应该使用正确的语法。

通过 DTD 验证的 XML 文档称为有效的 XML 文档，一个有效的 XML 文档应该遵守 DTD 的描述。

XML DTD 定义了 XML 文档中可用的合法元素。

XML Schema（XML 模式）是基于 XML 的 DTD 替代品，W3C 使得 DTD 和 Schema 可以相互替代。

XML 文档中发生错误将导致 XML 程序停止，所以语法正确时，XML 文档才能解析完成，即此文档是有效的 XML 文档。

15. DTD 语法

DTD 是“有效 XML 文档”的必需文件，通过 DTD 文件来定义文档中元素和标识的规则及相互关系。

（1）设置元素

元素是 XML 文档的基本组成部分，使用时需在 DTD 中定义一个元素，然后在 XML 文档中使用。元素在 DTD 中定义的语法格式为：

```
<!ELEMENT DESCRIPTION (#PCDATA, DEFINITION)*>
```

其中，“<!ELEMENT”是元素的声明，说明要定义的是一个元素，声明后面的“DESCRIPTION”是元素的名称，“(#PCDATA, DEFINITION)*>”则是该元素的使用规则。

（2）DTD 元素定义规则表

表 19-2 列出了 DTD 中元素定义的规则，DTD 中定义的规则说明了元素可以包含的内容以及相互关系。

表 19-2 DTD 中元素定义规则表

符 号	含 义	举 例
#PCDATA	包含字符或文本数据	<MYFILE(#PCDATA)> 元素 MYFILE 包含一个文本数据
#PCDATA, element-name	包含文本和其他子元素	<MYFILE(#PCDTATA,TITLE)>MYFILE 元素必须包含文本和 TITLE 子元素
,	使用逗号分隔排序	<MYFILE (TITLE ,AUTHOR ,EMAIL)>MYFILE 元素必须依次包含 TITLE、AUTHOR、EMAIL 三个子元素
	使用" "表示或者	<MYFILE (TITLE AUTHOR EMAIL)> MYFILE 元素必须包含 TITLE、AUTHOR 或者 EMAIL 子元素
name	只能使用一次	<MYFILE (TITLE)> MYFILE 元素必须包含 TITLE 子元素，而且只能使用一次
name?	使用一次或者不使用	<MYFILE (TITLE,AUTHOR?,EMAIL?)> MYFILE 元素必须包含 TITLE 子元素，而且只能使用一次；可以包含或者不包含 AUTHOR 和 EMAIL 子元素，但是如果使用，只能一次
name+	使用至少一次或多次	<MYFILE (TITLE+,AUTHOR?,EMAIL)> MYFILE 元素必须包含 TITLE 子元素，而且至少使用一次；接下来可以跟随 AUTHOR 子元素，也可以不跟；最后必须包含 EMAIL 子元素，而且只能使用一次
name*	使用一次，多次，或者根本不使用	<MYFILE (TITLE*)>MYFILE 元素可以包含一个、多个或者不包含 TITLE 子元素
()	设置组，可以嵌套	<MYFILE(#PCDATA TITLE)*>元素 MYFILE 包含一个或者更多的文本或者 TITLE 子元素。 <MYFILE((TITLE*, AUTHOR?, EMAIL)* COMMENT)> MYFILE 元素必须包含一些内容，内容或者是一个注释，或者是多个组，组里包含一个、多个或者没有 TITLE 子元素，接着是一个或者没有 AUTHOR 子元素，再接着是一个必需的 EMAIL 子元素

(3) DTD 定义举例说明

下面建立 myfile.xml 文件，然后在 myfile.dtd 中定义此文件的元素规则。

建立 myfile.xml 文件，文件内容如下：

```
<?xml version="1.0" encoding="GB2312"?>
<!DOCTYPE myfile SYSTEM "myfile.dtd">
<myfile>
<title>XML 轻松学习手册</title>
<author>ajie</author>
</myfile>
```

建立 myfile.dtd 文件，文件内容如下：

```
<!ELEMENT myfile (title, author)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
```

16. XML 编码

对 XML 编码的建议如下：

- ① 使用一种支持 Unicode 编码格式的编辑器。
- ② 确信知道自己正在使用哪种编码格式。
- ③ 在 XML 文档中使用属性声明设置编码格式。
- ④ XML 文档说明要以<?开始，以?>结束。
- ⑤ XML 文档编码定义格式如下：

```
<?xml version="1.0" encoding="ISO-8859-1"?>  
<?xml version="1.0" encoding="UTF-8"?>  
<?xml version="1.0" encoding="gb2312"?>
```

17. DOM 树

DOM (Document Object Model, 即文档对象模型) 树是将 XML 文件或 XML 字符串在内存中建立文档对象树，方便 XML 节点增、删、改、查操作。

使用 XML 时，一般需要首先将 XML 文档读入内存形成 DOM 树，使之结构化，以便进一步操作。图 19-2 画出了 DOM 树的节点种类及其层次关系，树中的每个 Document、Element、Text 和 Attr 都是 DOM 节点。

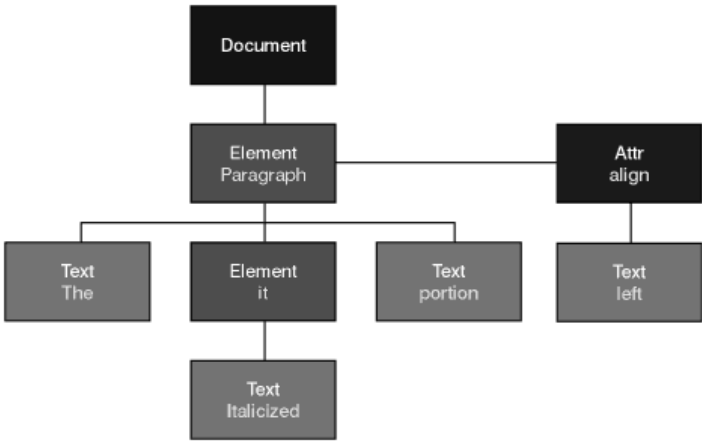


图 19-2 DOM 树层次图

使用 DOM 的基本规则为：不要使用 DOM 来遍历文档。只要可能，就使用 XPath 来查找节点或遍历文档。使用高级函数库使 DOM 更易于使用。

下面以 bookstore.xml 文件来说明 DOM 树，此文件内容如下：

```
<bookstore>  
<book category="CHILDREN">  
  <title lang="en">Harry Potter</title>
```

```
<author>J K. Rowling</author>
<year>2005</year>
<price>29.99</price>
</book>
</bookstore>
```

bookstore.xml 文件中的根元素是 bookstore，文档中的 book 元素都被包含在 bookstore 中，book 元素有 4 个子元素，分别为 title、author、year、price。bookstore.xml 形成 DOM 树，如图 19-3 所示。

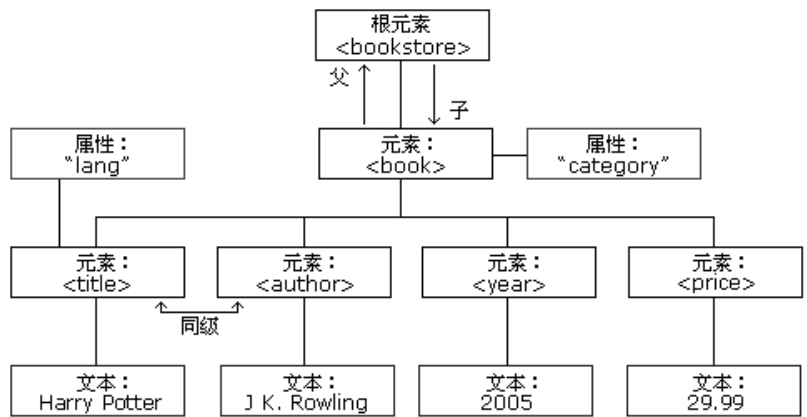


图 19-3 DOM 树实例图

18. XML 术语

- ① XHTML：可扩展 HTML（Extensible HTML）。
- ② XSL：可扩展样式表语言（Extensible Style Sheet Language）。
XSL 由三部分组成：XML 文档转换（XML Document Transformation，又叫 XSLT）、模式匹配语法（XPath）、格式化对象（XSL FO）。
- ③ XSLT：XML 转换语言（XML Transformation）。XSLT 是一种比 CSS 功能更强的语言，它可将 XML 文档转换成其他格式的文档。
- ④ Xpath：XML 匹配模式（XML Pattern Matching）。XPath 是一种用于标识 XML 文档各个部分的语言，这是一种为了 XSLT 和 XPointer 而设计出来的语言。
- ⑤ XPointer：XML 指针语言（XML Pointer Language）。XML 指针语言（The XML Pointer Language, XPointer），标识 XML 文档的内部结构，例如元素、属性、内容等。
- ⑥ XLink：XML 链接语言（XML Linking Language）。链接语言（The XML Linking Language, XLink），允许在不同的 XML 资源之间建立链接关系。
- ⑦ DTD：文档类型定义（Document Type Definition）。DTD 主要用于定义编写 XML 文档所使用的元素。
- ⑧ Namespaces：命名空间。XML 命名空间提供一种可以把元素、属性、命名和 URL 地址

引用相互关联的方法。

- ⑨ DOM: 文档对象模型 (Document Object Model)。DOM 定义了 XML 文档的接口、属性和方法。
- ⑩ SAX: XML 的简单 API (Simple API for XML)。SAX 是另一种读取和操作 XML 文档的编程接口 (与 DOM 类似)。
- ⑪ Schema: 中文称模式。与 DTD 不同, 它本身也是基于 XML 的。XML Schema 同时还支持名称空间, 能够定义比 DTD 更复杂的数据类型和结构。XML Schema 内置支持一系列的简单数据类型, 如字符串、小数和整数等, 还可以定义元素出现的次数。因此, XML Schema 更适合以数据为中心的文档。

19.3 XPath 语法

Xpath 和 XML 的关系相当于 SQL 语句和数据库的关系。本节详细描述 XPath 语法及其使用, XPath 表达式用得最多的还是对元素和属性的查找。

19.3.1 XPath 基本语法

XML Path 语言 (XPath) 表达式使用路径表示法 (像在 URL 中使用的一样) 来为 XML 文档的各部分寻址。表达式计算为生成节点集、布尔值、数字或字符串类型的对象。在 XPath 中, 有七种类型的节点: 元素、属性、文本、命名空间、处理指令、注释以及文档 (根) 节点。XML 文档是被作为节点树来对待的, 树的根被称为文档节点或者根节点。

1. URL 与 XPath 的比较

表 19-3 列出了 URL 与 XPath 的比较说明。

表 19-3 URL 与 XPath 的比较表

URL	XPath 表达式
由文件系统中的文件夹和文件组成的层次结构	由 XML 文档中的元素和其他节点组成的层次结构
每个级别具有唯一名称的文件, URL 总是标识单个文件	每个级别的元素名可能不是唯一的, XPath 表达式标识所有匹配的元素集
相对特定的文件夹 (称为“当前文件夹”) 进行计算	相对特定的节点 (称为表达式的“上下文”) 进行计算

2. 运算符和特殊字符

表 19-4 列出了 XPath 的运算符、特殊字符及其具体说明。

表 19-4 运算符和特殊字符表

符 号	含 义
/	子运算符, 即选择左侧集合的直接子级。此路径运算符出现在模式开头时, 表示应从根节点选择该子级
//	递归下降, 即在任意深度搜索指定元素。此路径运算符出现在模式开头时, 表示应从根节点递归下降
.	指示当前上下文

续表

符 号	含 义
..	当前上下文节点的父级
*	通配符。选择所有的元素，与元素名无关
@	属性。属性名的前缀
@*	属性通配符。选择所有的属性，与名称无关
:	命名空间分隔符。将命名空间前缀与元素名或属性名分隔
()	为运算分组，明确设置优先级
[]	应用筛选模式
[]	下标运算符。用于在集合中编制索引
+	执行加法
-	执行减法
div	根据 IEEE 754 执行浮点除法
*	执行乘法
mod	截断除法返回余数

3. 运算符优先级

表 19-5 列出了 XPath 的运算符优先级。

表 19-5 运算符优先级表

优 先 级	字 符	用 途
1	()	分组
2	[]	筛选器
3	/ //	路径运算
4	< 或者 <; <= 或者 <=; > 或者 >; >= 或者 >=;	比较
5	= !=	比较
6		联合
7	not()	布尔值非
8	and	布尔值与
9	or	布尔值或

4. XPath 路径符号表

表 19-6 列出了 XPath 的路径符号的含义，表 19-7 和表 19-8 列出了 XPath 路径符号的使用实例和具体说明。

表 19-6 XPath 路径符号表

符号	说 明	例 子	含 义
./	当前上下文	./author	以句点和正斜杠 (./) 作为前缀的表达式，明确使用当前上下文作为上下文
/	文档根	/bookstore	以正斜杠 (/) 为前缀的表达式使用文档树的根作为上下文

续表

符号	说 明	例 子	含 义
/*	根元素	/*	使用正斜杠后接星号（/*）的表达式将使用根元素作为上下文
//	递归下降	//author	使用双正斜杠（//）的表达式指示可以包括零个或多个层次结构级别的搜索
x/y	特定元素	bookstore/book	以元素名开头的表达式引用特定元素的查询，从当前上下文节点开始

表 19-7 路径运算符实例表

符 号	含 义
author/first-name	当前上下文节点的 <author> 元素中的所有 <first-name> 元素
bookstore//title	<bookstore> 元素中一级或多级深度的所有 <title> 元素（任意后代）。注意，此表达式与以下模式 bookstore/*/title 不同
bookstore//book/excerpt//emph	<book> 元素的 <excerpt> 子级中的任意位置和 <bookstore> 元素中的任意位置的所有 <emph> 元素
.//title	当前上下文中一级或多级深度的所有 <title> 元素。注意，本质上只有这种情况需要句点表示法

表 19-8 路径通配符表

符 号	含 义
author/*	<author> 元素的所有元素子级
book/*/last-name	所有作为 <book> 元素孙代的所有<last-name> 元素
/	当前上下文的所有孙级元素
my:book	my 命名空间中的 <book> 元素
my:*	my 命名空间中的所有元素

5. XPath 属性

XPath 使用@符号表示属性名。属性和子元素应公平对待，两种类型之间的功能应尽可能相当。属性不能包含子元素，所以，如果对属性应用路径运算符，将出现语法错误。此外，不能对属性应用索引，语法规则不能为属性定义任何顺序。表 19-9 列出了 XPath 属性的使用。

表 19-9 XPath 属性表

表 达 式	引 用
@style	当前元素上下文的 style 属性
price/@exchange	当前上下文内的 <price> 元素的 exchange 属性
book/@*style	所有 <book> 元素的 style 属性
@*	当前上下文节点的所有属性
@my:*	my 命名空间中的所有属性。不包括 my 命名空间中的元素的未限定属性

6. XPath 集合和筛选

XPath 查询返回的集合在定义的范围内保留文档顺序、层次结构和标识。也就是说，按照文档顺序返回元素集合。因为根据定义，属性不排序，所以为特定元素返回的属性不进行明确地排序。

具有特定标记名的所有元素集合使用标记名本身表示，限定方法可以通过使用句点和正斜杠

(./) 表明元素是从当前上下文中选择，但是默认情况下将使用当前上下文，不必明确说明。表 19-10 列出了 XPath 集合的使用。

表 19-10 XPath 集合表

表 达 式	引 用
./first-name	所有的 <first-name> 元素。注意，此表达式等效于后面的表达式
first-name	所有的 <first-name> 元素
author[1]	第一个 <author> 元素
author[first-name][3]	第三个具有 <first-name> 子级的 <author> 元素
x/y[1]	每个 <x> 内的第一个 <y>
(x/y)[1]	<x> 元素内 <y> 元素的整个集中的第一个 <y>
x[1]/y[1]	第一个 <x> 内的第一个 <y>
book[last()]	最后一个 <book> 元素
book/author[last()]	每个 <book> 元素内的最后一个 <author> 元素
(book/author)[last()]	<book> 元素内 <author> 元素的整个集中的最后一个 <author> 元素
(book/author)	所有来自当前上下文节点的任何 <book> 元素的子元素的 <author> 元素
author[(degree or award) and publication]	所有包含至少一个 <degree> 或 <award> 元素和至少一个 <publication> 元素的 <author> 元素

通过将筛选子句[pattern] 添加到集合中，可以对任何集合应用约束和分支，筛选器类似于 SQL WHERE 子句。筛选器中包含的模式称为“筛选模式”，筛选模式计算为布尔值，对集合中的每个元素进行测试。集合中所有未通过筛选模式测试的元素将从结果集合中省略。表 19-11 列出了 XPath 集合筛选模式表。

表 19-11 XPath 集合筛选模式表

表 达 式	引 用
book[excerpt]	所有包含至少一个 <excerpt> 元素的 <book> 元素
book[excerpt]/title	<book> 元素内所有包含至少一个 <excerpt> 元素的 <title> 元素
book[excerpt]/author[degree]	所有包含至少一个 <degree> 元素并且再包含至少一个 <excerpt> 元素的 <book> 元素之内的 <author> 元素
book[author/degree]	所有包含至少一个<author> 元素并且这个元素有至少一个 <degree> 子元素的 <book> 元素
book[excerpt][title]	所有包含至少一个 <excerpt> 元素和至少一个 <title> 元素的 <book> 元素

7. 布尔、比较和集表达式

表 19-12 列出了 XPath 的运算符种类及其具体说明，表 19-13 列出了 XPath 布尔表达式的使用实例及其具体说明。

表 19-12 布尔、比较和集运算符表

运 算 符	说 明
and	逻辑与
or	逻辑或

续表

运 算 符	说 明
not()	非
=	相等
!=	不相等
<	小于
>=	小于或等于
>	大于
<=	大于或等于
	集运算，返回两个节点集的联合

表 19-13 XPath 布尔表达式使用实例表

布尔表达式	说 明
author[degree and award]	至少包含一个 <degree> 元素和至少一个 <award> 元素的所有 <author> 元素
author[(degree or award) and publication]	至少包含一个 <degree> 或 <award> 元素以及至少一个 <publication> 元素的所有 <author> 元素
author[degree and not(publication)]	至少包含一个 <degree> 元素但是不包含 <publication> 元素的所有 <author> 元素
author[not(degree or award) and publication]	至少包含一个 <publication> 元素但是不包含任何 <degree> 元素或 <award> 元素的所有 <author> 元素

8. 对象比较说明

要在 XPath 中比较两个对象，可以使用“=”测试是否相等，也可以使用“!=”测试是否不相等。

对于比较运算，必须正好提供两个操作数，比较的过程是计算每个操作数，然后根据需要将操作数转换为相同的类型，最后完成比较。

XML 的所有元素和属性都是字符串，在进行数字比较时会自动被强制转换为整数，在比较运算期间，文本数值会被强制转换为 long 或 double 类型，转换规则如下：

- ① 如果至少有一个操作数为布尔值，每个操作数必须先转换为布尔值。
- ② 否则，如果至少有一个操作数为数字，每个操作数必须先转换为数字。
- ③ 否则，如果至少有一个操作数为日期，每个操作数必须先转换为日期。
- ④ 否则，两个操作数都先转换为字符串。

表 19-14 列出了对象间的比较方法。

表 19-14 对象比较方法表

文本类型	比 较	示 例
String	text(lvalue) op text(rvalue)	a < GGG
Integer	(long) lvalue op (long) rvalue	a < 3
Real	(double) lvalue op (double) rvalue	a < 3.1

单引号或双引号可以作为表达式中字符串的分隔符，这样更容易从脚本语言内部构造和传递模式。表 19-15 列出了 XPath 中字符串的使用，字符串使用时需要用单引号或双引号引起来。

表 19-15 XPath 中字符串使用实例表

表 达 式	说 明
author[last-name = "Bob"]	至少包含一个带有 Bob 值的 <last-name> 元素的所有 <author> 元素
author[last-name[1] = "Bob"]	其第一个 <last-name> 子元素具有 Bob 值的所有 <author> 元素
author/degree[@from != "Harvard"]	包含 from 属性不等于“Harvard”的<degree>元素的所有<author>元素
author[last-name = /editor/last-name]	所包含的<last-name>元素与根元素下<editor>元素内部的<last-name>元素相同的所有<author>元素
author[. = "Matthew Bob"]	所有字符串值为 Matthew Bob 的 <author> 元素

可使用“<”、“<=”、“>”和“>=”运算符分别表示小于、小于或等于、大于以及大于或等于。对象比较运算返回的是布尔值，表 19-16 列出了对象比较运算实例及其说明。

表 19-16 对象比较运算实例表

表 达 式	说 明
author[last-name = "Bob" and price > 50]	所包含的 <last-name> 元素带有 Bob 值、<price> 元素的值大于 50 的所有 <author> 元素
degree[@from != "Harvard"]	所有 from 属性不等于“Harvard”<degree> 的元素
book[position() <= 3]	XML 文件中的头 3 个<book> 元素（1，2，3）

9. XPath 集运算

Union（|）运算符返回两个操作数的联合，操作数必须是节点集。表 19-17 列出了 XPath 集运算实例及其说明。

表 19-17 XPath 集运算表

表 达 式	说 明
first-name last-name	包含当前上下文中的 <first-name> 和 <last-name> 元素的节点集
(bookstore/book bookstore/magazine)	包含 <bookstore> 元素中的 <book> 或 <magazine> 元素的节点集
book book/author	包含 <book> 元素中的所有<book> 元素和所有<author> 元素的节点集
(book magazine)/price	包含 <book> 或 <magazine> 元素的所有 <price> 元素的节点集

19.3.2 XPath 位置路径

上文介绍了XPath基本语法，说明XPath每个语法单元的语法。而XPath位置路径则是介绍XPath表达式的整体语法，是上文XPath基本语法的综合运用。位置路径由轴、节点测试和谓词三部分组成，定位步骤说明了位置路径的定位方法。

1. 位置路径

位置路径是一种 XPath 表达式，用于选择相对于上下文节点的一组节点。计算位置路径表达

式所得到的节点集将包含位置路径指定的节点。位置路径可以以递归方式包含表达式，用来筛选节点集。

在语法上，位置路径由一个或多个定位步骤组成，每个步骤通过斜杠 (/) 分隔。其语法形式如下：

```
locationstep/locationstep/locationstep
```

每个定位步骤依次选择相对于上下文节点（即上一个定位步骤所选择的节点）的一组节点。通过这种方式表示的位置路径称为相对位置路径。

位置路径还有一种定位方法，称为绝对位置路径，其路径从根元素开始，语法形式如下：

```
/locationstep/locationstep/locationstep
```

在位置路径中，定位步骤从左到右进行计算，最左侧的定位步骤选择一组相对于上下文节点的节点。然后，这些节点成为新的上下文节点，用于处理下一个定位步骤，通过这种方法直到所有的定位步骤处理完毕。

位置路径可以缩写，也可以不缩写。在不缩写的位置路径中，定位步骤采用以下语法：

```
axis::node-test[predicate]
```

在此语法中，“axis”指定定位步骤所选择的节点与上下文节点的关系。“node-test”指定定位步骤选择节点的节点类型和扩展名称。“predicate”为谓词，是一个筛选表达式，进一步精确定位步骤中的节点选择，此谓词是可选的。表 19-18 列出了 XPath 不缩写的位置路径用法，表 19-19 列出了 XPath 缩写的位置路径用法，表 19-20 列出了两者的对照关系。

表 19-18 不缩写位置路径表

不缩写的位置路径	说 明
child::para[last()]	选择上下文节点的最后一个 <para> 元素
parent::para	选择属于上下文节点父级的 <para> 元素
child::text()	选择上下文节点的所有文本节点子级
child::div/child::para	选择属于上下文节点子级的 <div> 元素的 <para> 子元素

表 19-19 缩写位置路径表

缩写的位置路径	说 明
para	选择上下文节点的 <para> 元素
../para	选择属于上下文节点父级的 <para> 元素
text()	选择上下文节点的所有文本节点子级
./div/para	选择上下文节点的 <div> 元素子级的 <para> 元素子级

表 19-20 不缩写位置路径与缩写位置路径对照表

不 缩 写	缩 写
child::*	*
attribute::*	@*
/descendant-or-self::node()	//

续表

不 缩 写	缩 写
self::node()	.
parent::node()	..

2. 定位步骤

定位步骤相对于上下文节点选择一组节点（节点集）。
定位步骤分三个部分：可选轴、节点测试和可选谓词。定位步骤的语法是轴名后接双冒号，然后接节点测试，最后是零个或多个谓词（每个谓词都放在方括号中）。此语法最基本的形式如下：

```
axis::nodetest[predicate]
```

其中，axis 为轴，指定上下文节点与定位步骤所选节点之间的树形关系，也就是说，轴指示定位步骤从上下文节点开始的常规方向。在定位步骤中，轴是可选项，如果省略，轴默认为 child::。

nodetest 为节点测试，指定定位步骤最初选择节点的节点类型或扩展名称。基本上，节点测试指定轴上的所有节点中哪些节点视为定位步骤的候选（即潜在）匹配项。

predicate 为谓词，使用 XPath 表达式（要满足的条件）进一步精选定位步骤所选的节点集。谓词是一个过滤器，通过指定选择标准来进一步精选候选节点列表。谓词是可选项，如果没有谓词，定位步骤中就没有方括号（[]）。

选择节点是根据轴和节点测试之间的关系生成初始节点集，然后依次通过每个谓词筛选该初始节点集，从而确定定位步骤所选的节点集。初始节点集由满足下列两个条件的节点组成：

- ① 节点与轴指定的上下文节点有关系。
- ② 节点具有节点测试指定的节点类型和扩展名称。

XPath 使用定位步骤中的第一个谓词筛选初始节点集，以生成新的节点集；然后，XPath 使用第二个谓词筛选第一个谓词生成的节点集。此筛选过程重复进行，直到 XPath 计算完成所有的谓词。应用了所有的谓词之后生成的节点集就是定位步骤所选的节点集。

3. 轴

位置路径使用轴来指定定位步骤所选的节点与上下文节点之间的关系。轴分为正轴、反轴和其他轴，正轴会按照正方向浏览 XML 树，反轴则反方向浏览 XML 树。

正轴是包含上下文节点或上下文节点之后的节点的轴，如 child::、descendant::、descendant-or-self::、following::和 following-sibling::轴是正轴。这些正轴从第一个位置开始按文档顺序为节点集中的节点编号。

反轴是包含上下文节点或上下文节点之前的节点的轴。如 ancestor::、ancestor-or-self::、preceding::和 preceding-sibling::轴是反轴。这些反轴按文档从第一个位置

开始顺序相反的顺序为节点集的节点编号。

对于其他轴，`self::`和 `parent::`轴返回单个节点，因此，指定正轴或反轴这两个轴没有意义。`attribute::`和 `namespaces::`轴没有定义顺序，所以，也没有正轴和反轴。

表 19-21 列出了 XPath 的轴类型及其具体说明。

表 19-21 XPath 轴表

轴 类 型	说 明
<code>ancestor::</code>	上下文节点的上级。上下文节点的上级由上下文节点的父级和父级的父级等组成。因此， <code>ancestor::</code> 轴总是包括根节点，除非上下文节点就是根节点
<code>ancestor-or-self::</code>	上下文节点及其上级。 <code>ancestor-or-self::</code> 轴总是包括根节点
<code>attribute::</code>	上下文节点的属性。除非上下文节点为元素，否则，此轴将是空的
<code>child::</code>	上下文节点的子级。子级是树中上下文节点以下紧邻的任何节点。但是，属性节点或命名空间节点均不属于上下文节点的子级
<code>descendant::</code>	上下文节点的子代。子代是子级或子级的子级，因此， <code>descendant::</code> 轴永远不会包含属性节点或命名空间节点
<code>descendant-or-self::</code>	上下文节点及其子代
<code>following::</code>	树中在上下文节点之后的所有节点，除子代、属性节点和命名空间节点之外
<code>following-sibling::</code>	上下文节点的所有后续同辈。 <code>following-sibling::</code> 轴只标识出现在树中上下文节点之后的父节点子级。该轴不包括所有出现在上下文节点之前的其他子级。如果上下文节点是属性节点或命名空间节点， <code>following-sibling::</code> 轴是空的
<code>namespace::</code>	上下文节点的命名空间节点。每个处于上下文节点范围内的命名空间都有一个命名空间节点。除非上下文节点为元素，否则，此轴将是空的
<code>parent::</code>	上下文节点的父级（如果有）。父级是树中上下文节点以上紧邻的节点
<code>preceding::</code>	树中在上下文节点之前的所有节点，除了上级、属性节点和命名空间节点之外。有一种方法是将 <code>preceding</code> 轴看做是内容全部出现在上下文节点开始之前的所有节点
<code>preceding-sibling::</code>	上下文节点的所有前接同辈。 <code>preceding-sibling::</code> 轴只标识出现在树中上下文节点之前的父节点子级。该轴不包括所有出现在上下文节点之后的其他子级。如果上下文节点是属性节点或命名空间节点， <code>preceding-sibling::</code> 轴是空的
<code>self::</code>	只是上下文节点本身

4. 节点测试

节点测试是 XPath 定位步骤唯一必选的部分。因此，了解节点测试对于成功使用 XPath 表达式至关重要。常见的节点测试类型有三种，分别为名称测试、节点类型测试和确定目标的处理指令测试。

在 XPath 中，主要有三种节点类型，表 19-22 列出了这三种节点类型及具体说明。

表 19-22 XPath 节点类型表

轴	主要节点类型
除 <code>attribute::</code> 轴或 <code>namespace::</code> 轴以外的任何轴（即可以包含元素的轴）	元素
属性	属性
命名空间	命名空间

(1) 名称测试

名称测试是最常见的一种节点测试形式，可以明确地指定要选择节点的名称。由于文档树可能包含同名的不同节点类型，名称测试需要根据轴和名称两项要素完成。

名称测试中指定的名称可能属于以下三种类型之一：星号 (*)、QName 或表达式 NCName:*。表 19-23 列出了每种类型的名称如何与指定的轴配合使用，以找到特定的节点集。

表 19-23 名称测试表

名 称	返 回	示 例
* (星号)	对于任何主要节点类型的节点，返回 True	ancestor::* 选择上下文节点的所有上级 attribute::* 选择上下文节点的所有属性 namespace::* 选择上下文节点的所有名称属性
QName	对于任何扩展名称等于 QName 指定的扩展名称的主要类型节点，返回 True	child::para 选择所有属于上下文节点子级的 <para> 元素节点。如果上下文节点没有 <para> 子级，则选择空节点集
NCName:*	对于任何扩展名称包含 NCName 展开到的命名空间 URI 的主要类型节点（与本地名称无关），返回 True	child::ns:* 选择带 ns 前缀的命名空间中的所有子元素节点

如果节点测试为 QName，XPath 必须先根据 XML 文件中的上下文命名空间声明展开 QName。如果 QName 没有前缀，XPath 将查找本地名称与给定 QName 匹配的节点。

如果节点测试为 NCName:*，XPath 展开 NCName 的方式与展开 QName 的方式相同。如果 NCName 部分与文件的上下文命名空间声明中包含的任何前缀均不对应，NCName:* 测试会报错。

(2) 节点类型测试

要选择除元素节点以外（或包括元素节点）的节点类型，则需使用节点类型测试。使用节点类型测试的作用是重写给定轴的主要节点类型。例如，descendant::text() 找到上下文节点以下的所有文本节点。

有四种节点类型测试，如表 19-24 所示。

表 19-24 节点类型测试表

名 称	返 回	示 例
comment()	对注释节点返回 true	following::comment() 选择所有出现在上下文节点之后的注释节点
node()	对任何类型的节点返回 true	preceding::node() 选择所有出现在上下文节点之前的节点
processing-instruction()	对处理指令节点返回 true	self::processing instruction() 选择上下文节点中的所有处理指令节点
text()	对文本节点返回 true	child::text() 选择属于上下文节点子级的文本节点

(3) 指定目标的处理指令测试

节点测试还可以是指定目标的处理指令。此类节点测试的语法如下：

```
processing-instruction("target")
```

通过指定目标处理指令测试找到所有与该目标匹配的处理指令节点。例如，以下节点测试的目的是找到文档中所有指定 XSLT 文件中处理指令的节点。

```
/child::processing-instruction("xml-stylesheet")
```

5. 谓词

谓词是一个 XPath 表达式，用于针对某个轴筛选的节点集，并生成一个新的节点集，此筛选过程包括按顺序针对节点集中的每个节点计算该谓词。每次针对节点计算该谓词时，上下文节点是当前计算的节点，上下文位置是上下文节点在节点集中的位置。

谓词表达式计算结果有两种，即数字或布尔值。如果谓词计算结果为数字，XPath 将该数字与上下文节点的上下文位置进行比较，如果数字和位置相同（即上下文节点处于树中相应的位置），此上下文节点将包含在新的节点集中，否则，此上下文节点将排除在新节点集之外。

如果谓词表达式计算结果不为数字，XPath 将使用 boolean 函数将结果转换为布尔值。例如，谓词[genre='Computer']计算为一个节点集，如果上下文节点有一个内容为 Computer 子级的 genre 元素，此谓词就计算为 true，并且该上下文节点被包括在新节点集内，否则，此上下文节点将排除在新节点集之外。

另外要说明的是，数字谓词[x]等效于布尔谓词[position()=x]。

表 19-25 列出了 XPath 中谓词使用实例的具体说明。

表 19-25 谓词使用实例表

定位步骤	说 明
child::*[position()=1]	定位上下文节点的第一个子节点
ancestor-or-self::book[@catdate="2000-12-31"]	只要涉及的元素具有带“2000-12-31”值的 catdate 属性，便定位上下文节点的任何<book>子级的上级以及<book>子级本身
//parent::node()[name()='book'] descendant::node()[name()='author']	定位文档中任何父节点名为“book”的节点或任何从名为“author”的上下文节点下降的节点

6. 位置路径使用示例

表 19-26 列出了位置路径的使用方法及其具体说明。

表 19-26 位置路径使用示例表

路 径	说 明
child::node()	选择上下文节点的所有子级，无论属于哪种节点类型
attribute::name	选择上下文节点的 name 属性
attribute::*	选择上下文节点的所有属性
descendant::para	选择上下文节点的 <para> 元素后代
ancestor::div	选择上下文节点的所有 <div> 上级
ancestor-or-self::div	选择上下文节点的 <div> 上级，如果上下文节点是一个 <div> 元素，则也选择该上下文节点

续表

路 径	说 明
descendant-or-self::para	选择上下文节点的 <para> 元素后代，如果上下文节点是一个 <para> 元素，则也选择该上下文节点
self::para	如果上下文节点是一个 <para> 元素，则选择该上下文节点，否则什么也不选
child::chapter/descendant::para	选择上下文节点的 <chapter> 元素子级的 <para> 元素后代
child::*/child::para	选择上下文节点的所有 <para> 孙级
/	选择文档根（总是文档元素的父级）
/descendant::para	选择与上下文节点同在一个文档中的所有 <para> 元素
/descendant::olist/child::item	选择具有 <olist> 父级并且与上下文节点同在一个文档中的所有 <item> 元素
child::para[position()=1]	选择上下文节点的第一个 <para> 子级
child::para[position()=last()]	选择上下文节点的最后一个 <para> 子级
child::para[position()=last()-1]	选择上下文节点的倒数第二个 <para> 子级
child::para[position()>1]	选择上下文节点的所有 <para> 子级，但上下文节点的第一个 <para> 子级除外
/descendant::figure[position()=42]	选择文档中第 42 个 <figure> 元素
/child::doc/child::chapter[position()=5]/child::section[position()=2]	选择 <doc> 文档元素的第 5 个 <chapter> 元素中包含的第 2 个 <section> 元素
child::para[attribute::type="warning"]	选择上下文节点中 type 属性值为“warning”的所有 <para> 子级
child::para[attribute::type="warning"][position()=5]	选择上下文节点中 type 属性值为“warning”的第 5 个 <para> 子级
child::para[position()=5][attribute::type="warning"]	如果上下文节点的第 5 个 <para> 子级的 type 属性值为“warning”，则选择该子级
child::chapter[child::title="Introduction"]	选择上下文节点中具有一个或多个字符串值等于“Introduction”的 <title> 子级的 <chapter> 子级
child::chapter[child::title]	选择上下文节点中具有一个或多个 <title> 子级的 <chapter> 子级
child::*[self::chapter or self::appendix]	选择上下文节点的 <chapter> 和 <appendix> 子级
child::*[self::chapter or self::appendix][position()=last()]	选择上下文节点的最后一个 <chapter> 或 <appendix> 子级

19.3.3 XPath 示例

1. XML 示例文件

XML 示例文件 inventory.xml 的内容如下：

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="myfile.xsl" ?>
<bookstore specialty="novel">
  <book style="autobiography">
    <author>
```



```

    <first-name>Joe</first-name>
    <last-name>Bob</last-name>
    <award>Trenton Literary Review Honorable Mention</award>
  </author>
  <price>12</price>
</book>
<book style="textbook">
  <author>
    <first-name>Mary</first-name>
    <last-name>Bob</last-name>
    <publication>Selected Short Stories of
      <first-name>Mary</first-name>
      <last-name>Bob</last-name>
    </publication>
  </author>
  <editor>
    <first-name>Britney</first-name>
    <last-name>Bob</last-name>
  </editor>
  <price>55</price>
</book>
<magazine style="glossy" frequency="monthly">
  <price>2.50</price>
  <subscription price="24" per="year"/>
</magazine>
<book style="novel" id="myfave">
  <author>
    <first-name>Toni</first-name>
    <last-name>Bob</last-name>
    <degree from="Trenton U">B.A.</degree>
    <degree from="Harvard">Ph.D.</degree>
    <award>Pulitzer</award>
    <publication>Still in Trenton</publication>
    <publication>Trenton Forever</publication>
  </author>
  <price intl="Canada" exchange="0.7">6.50</price>
  <excerpt>
    <p>It was a dark and stormy night.</p>
    <p>But then all nights in Trenton seem dark and
    stormy to someone who has gone through what
    <emph>I</emph> have.</p>
    <definition-list>
      <term>Trenton</term>
      <definition>misery</definition>
    </definition-list>
  </excerpt>
</book>
<my:book xmlns:my="uri:mynamespace" style="leather" price="29.50">
  <my:title>Who's Who in Trenton</my:title>
  <my:author>Robert Bob</my:author>
</my:book>
</bookstore>

```

2. XPath 操作示例

对上述示例文件 `inventory.xml`，表 19-27 列出了对其进行的各种操作方法，基本涵盖了

前面表中的各种操作方法，读者可以在实际工作中模仿使用。

表 19-27 X Path 操作示例表

表 达 式	含 义
./author	当前上下文中的所有 <author> 元素。注意，此表达式等效于下一行中的表达式
author	当前上下文中的所有 <author> 元素
first.name	当前上下文中的所有 <first.name> 元素
/bookstore	此文档的文档元素 <bookstore>
//author	文档中的所有 <author> 元素
book[/bookstore/@specialty=@style]	style 属性值等于文档根处 <bookstore> 元素的 specialty 属性值的所有 <book> 元素
author/first-name	作为 <author> 元素子级的所有 <first-name> 元素
bookstore//title	<bookstore> 元素中一级或多级深度的所有 <title> 元素（任意后代）。注意，此表达式不同于下一行中的表达式
bookstore/*/title	作为 <bookstore> 元素的孙代的所有 <title> 元素
bookstore//book/excerpt//emph	位于 <book> 元素的 <excerpt> 子级内任意位置和位于 <bookstore> 元素内任意位置的所有 <emph> 元素
../title	当前上下文中一级或多级深度的所有 <title> 元素。注意，本质上只有这种情况需要句点表示法
author/*	作为 <author> 元素子级的所有元素
book/*/last-name	作为 <book> 元素孙级的所有<last-name>元素
/	当前上下文的所有孙级元素
*[@specialty]	具有 specialty 属性的所有元素
@style	当前上下文的 style 属性
price/@exchange	当前上下文中 <price> 元素的 exchange 属性
price/@exchange/total	返回空节点集，因为属性不包含元素子级。XML 路径语言（XPath）语法允许使用此表达式，但是严格意义上讲无效
book[@style]	当前上下文的具有 style 属性的所有 <book>元素
book/@style	当前上下文的所有<book> 元素的 style 属性
@*	当前元素上下文的所有属性
./first-name	当前上下文节点中的所有 <first-name> 元素。注意，这等效于下一行中的表达式
first-name	当前上下文节点中的所有 <first-name> 元素
author[1]	当前上下文节点中的第一个 <author> 元素
author[first-name][3]	具有 <first-name> 子级的第三个 <author> 元素
my:book	my 命名空间中的 <book> 元素
my:*	my 命名空间中的所有元素
@my:*	my 命名空间中的所有属性（不包括 my 命名空间中的元素的未限定属性）
book[last()]	当前上下文节点的最后一个 <book> 元素
book/author[last()]	当前上下文节点的每个 <book> 元素的最后一个 <author> 子级
(book/author)[last()]	当前上下文节点的 <book> 元素的整个 <author> 子级集合中的最后一个 <author> 元素

续表

表 达 式	含 义
book[excerpt]	包含至少一个 <excerpt> 元素子级的所有 <book> 元素
book[excerpt]/title	作为 <book> 元素子级, 同时包含至少一个 <excerpt> 元素子级的所有 <title> 元素
book[excerpt]/author[degree]	包含至少一个 <degree> 元素子级, 并作为同样包含至少一个 <excerpt> 元素的 <book> 元素的子级的所有 <author> 元素
book[author/degree]	包含 <author> 子级, 这些子级又包含至少一个 <degree> 子级的所有 <book> 元素
author[degree][award]	包含至少一个 <degree> 元素子级和至少一个 <award> 元素子级的所有 <author> 元素
author[degree and award]	包含至少一个 <degree> 元素子级和至少一个 <award> 元素子级的所有 <author> 元素
author[(degree or award) and publication]	包含至少一个 <degree> 或 <award> 和至少一个 <publication> 作为子级的所有 <author> 元素
author[degree and not (publication)]	包含至少一个 <degree> 元素子级, 但包含 <publication> 元素子级的所有 <author> 元素
author[not(degree or award) and publication]	包含至少一个 <publication> 元素子级, 但不包含 <degree> 或 <award> 元素子级的所有 <author> 元素
author[last-name = "Bob"]	包含至少一个值为 Bob 的 <last-name> 元素子级的所有 <author> 元素
author[last-name[1] = "Bob"]	第一个 <last-name> 子元素的值为 Bob 的所有 <author> 元素。注意, 这等效于下一行中的表达式
author[last-name [position()=1] = "Bob"]	第一个 <last-name> 子元素的值为 Bob 的所有 <author> 元素
degree[@from != "Harvard"]	from 属性不等于 “Harvard” 的所有 <degree> 元素
author[. = "Matthew Bob"]	值为 Matthew Bob 的所有 <author> 元素
author[last-name = "Bob" and ../price > 50]	包含值为 Bob 的 <last-name> 子元素和值大于 50 的 <price> 同级元素的所有 <author> 元素
book[position() <= 3]	前三个 book 元素 (1、2、3)
author[not(last-name = "Bob")]	不包含值为 Bob 的 <last-name> 子元素的所有 <author> 元素
author[first-name = "Bob"]	至少有一个值为 Bob 的<first-name>子级的所有<author> 元素
author[* = "Bob"]	所有包含任何值为 Bob 的子元素的 author 元素
author[last-name = "Bob" and first-name = "Joe"]	具有值为 Bob 的 <last-name> 子元素和值为 Joe 的<first-name> 子元素的所有 <author> 元素
price[@intl = "Canada"]	上下文节点中 intl 属性等于 “Canada” 的所有 <price> 元素
degree[position() < 3]	作为上下文节点子级的前两个 <degree> 元素
p/text()[2]	上下文节点中每个 <p> 元素的第二个文本节点
ancestor::book[1]	上下文节点最近的 <book> 上级
ancestor::book[author][1]	上下文节点最近的 <book> 上级, 并且此 <book> 元素具有 <author> 元素作为其子级
ancestor::author[parent::book][1]	当前上下文最近的 <author> 上级, 并且此 <author> 元素是 <book> 元素的子级

第 20 章

libxml编程

libxml 是一个 XML C 语言版的解析器，也是一个基于 MIT License 的免费开源软件，同时也是 Linux 平台最常使用的 XML 解析器。

20.1 libxml 编程基础

本节主要介绍 libxml 的安装、libxml 主要的数据类型、libxml 的主要函数和 XML 常见操作。其中，XML 常见操作包括创建 XML 文档、解析 XML 文档、修改 XML 文档、使用 XPath 查找 XML 文档、利用 iconv 工具实现字符集转码。

20.1.1 libxml 的安装

在安装系统的时候，如果选中了 libxml 开发库，系统会默认安装 libxml。如果没有安装，可以按如下步骤进行手工安装。

① 从 xmlsoft 站点或 [ftp\(ftp.xmlsoft.org\)](ftp://ftp.xmlsoft.org) 站点下载 libxml 压缩包 (libxml2-xxxx.tar.gz)。

② 利用下列命令对压缩包进行解压缩。

```
tar xvzf libxml2-xxxx.tar.gz
```

③ 进入解压缩后的文件夹中运行如下命令完成安装。

```
./configure  
make  
make install
```

也可以使用 `./configure --prefix=$HOME/xml2lib` 指定安装目录，如果不指定目录，默认安装在系统目录 `“/usr/local/include/libxml2”` 下。

安装完成后，就可以使用简单的代码解析 XML 文件，包括本地和远程的文件。但是在编码上可能有一些问题，libxml 默认只支持 UTF-8 编码，无论输入/输出的内容是什么，都只能是 UTF-8

编码，如果需要输出 GB2312 或者其他编码的文字或字符串，就需要用 iconv 工具库来做转码，安装 iconv 工具库的方法如下：

① 下载 libiconv 压缩包（如 libiconv-1.11.tar.gz）。

② 利用下列命令对压缩包进行解压缩。

```
tar xvzf libiconv-1.11.tar.gz
```

③ 进入解压缩后的文件目录，运行如下命令完成安装。

```
./configure  
make  
make install
```

20.1.2 libxml 主要的数据类型

下面介绍的是 libxml 主要的数据类型，对于应用编程来说，这些数据类型是需要了解和掌握的。

1. 内部字符类型 xmlChar

xmlChar 是 libxml2 中的字符类型，库中所有的字符、字符串都是基于这个数据类型的。它在 xmlstring.h 中的定义说明如下：

```
typedef unsigned char xmlChar;
```

使用 unsigned char 作为内部字符格式是考虑到它能很好地适应 UTF-8 编码，而 UTF-8 编码正是 libxml2 的内部编码，其他格式的编码要转换为这个编码才能在 libxml2 中使用。

xmlChar * 常在 libxml2 中作为字符串指针类型，很多函数会返回一个动态分配内存的 xmlChar * 变量，使用这样的函数时，需要手工删除内存。

2. xmlChar 相关函数

如同标准 C 中的 char 类型一样，xmlChar 也有动态内存分配、字符串操作等相关函数。例如，xmlMalloc 是动态分配内存的函数，xmlFree 是配套的释放内存函数，xmlStrcmp 是字符串比较函数等。基本上，xmlChar 字符串相关的函数都在 xmlstring.h 中定义，而动态内存分配函数在 xmlmemory.h 头文件中定义。

3. xmlChar* 与其他类型之间的转换

在实际编程中，总是需要在 xmlChar * 和 char * 之间进行强制类型转换的，所以定义了一个宏 BAD_CAST，其定义如下：

```
#define BAD_CAST (xmlChar *)
```

4. XML 中常用的重定义

在 XML 程序中，经常会看到 xmlChildrenNode 这个名称，其实这个名称是定义在 tree.h

中的重定义。其重定义如下：

```
#define xmlChildrenNode children
```

5. 文档类型 xmlDoc、指针 xmlDocPtr

xmlDoc 是一个结构 (struct)，保存了 XML 的一个相关信息，例如，文件名、文档类型、子节点等，xmlDocPtr 等于 xmlDoc *。与文档指针相关的函数有如下几个。

- xmlDocNewDoc 函数创建一个新的文档指针。
- xmlDocParseFile 函数以默认方式读入一个 UTF-8 格式的文档，并返回文档指针。
- xmlDocReadFile 函数读入一个带有某种编码的 xml 文档，并返回文档指针。
- xmlDocFreeDoc 释放文档指针。特别注意，当调用 xmlDocFreeDoc 时，该文档包含的所有节点内存都会被释放，所以，一般不需要手工调用 xmlDocFreeNode 或者 xmlDocFreeNodeList 来释放动态分配的节点内存，除非把该节点从文档中移除。一般来说，一个文档中所有的节点都应该动态分配，然后加入文档，最后调用 xmlDocFreeDoc 一次释放所有节点申请的动态内存，这也是为什么我们在程序中很少看见 xmlDocNodeFree 的原因。
- xmlDocSaveFile 将文档以默认方式存入一个文件。
- xmlDocSaveFormatFileEnc 可将文档以某种编码格式存入一个文件中。

6. 节点类型 xmlNode、指针 xmlDocPtr

节点是 XML 中最重要的元素，xmlNode 代表 XML 文档中的一个节点，实现为一个结构 (struct)，此结构内容很丰富，也很重要，它定义在 tree.h 中，具体说明如下：

```
typedef struct _xmlNode xmlNode;
typedef xmlNode *xmlNodePtr;
struct _xmlNode {
    void                *_private; /* 应用数据 */
    xmlElementType     type;      /* 元素类型，必须是 2 */
    const xmlChar       *name;     /* 节点名称 */
    struct _xmlNode     *children; /* 子节点指针 */
    struct _xmlNode     *last;     /* 最后一个子节点指针 */
    struct _xmlNode     *parent;   /* 父节点指针 */
    struct _xmlNode     *next;     /* 下一个兄弟节点指针 */
    struct _xmlNode     *prev;     /* 上一个兄弟节点指针 */
    struct _xmlDoc      *doc;      /* 节点文档指针 */
    /* End of common part */
    xmlNs               *ns;       /* 节点关联的名字空间指针 */
    xmlChar              *content; /* 节点文字内容指针 */
    struct _xmlAttr     *properties; /* 节点属性列表指针 */
    xmlNs               *nsDef;    /* 节点定义的名称空间指针 */
    void                *psvi;    /* PSVI 类型信息 */
    unsigned short      line;      /* 行号 */
    unsigned short      extra;     /* XPath 的附加数据 */
};
```

可以看到，节点之间是以链表和树两种方式同时组织起来的，`next` 和 `prev` 指针可以组成链表，而 `parent` 和 `children` 可以组织为树。同时，此结构还有以下重要成员：

- `content`：节点中的文字内容。
- `doc`：节点所属文档。
- `name`：节点名字。
- `ns`：节点的名字空间。
- `properties`：节点属性列表。

XML 文档操作的根本原理就是在节点之间移动、查询节点的各项信息，并进行增加、删除、修改等操作。

`xmlDocSetRootElement` 函数可以将一个节点设置为某个文档的根节点，这是将文档与节点连接起来的重要手段。当有了根节点以后，所有的子节点就可以依次连接上根节点，从而组织成为一个 XML 树。

7. XML 属性

XML 属性也是编程中经常用到的结构，其定义如下：

```
struct _xmlAttr {
    void *_private; /* 应用数据指针 */
    xmlElementType type; /* 属性类型 */
    const xmlChar * name; /* 属性名称指针 */
    struct _xmlNode * children; /* 属性内容指针 */
    struct _xmlNode * last; /* 需等于 NULL 值 */
    struct _xmlNode * parent; /* 父节点指针 */
    struct _xmlAttr * next; /* 下一个兄弟节点指针 */
    struct _xmlAttr * prev; /* 上一个兄弟节点指针 */
    struct _xmlDoc * doc; /* 节点文档指针 */
    xmlNs * ns; /* 属性名字空间指针 */
    xmlAttributeType atype; /* 属性类型验证 */
    void * psvi; /* PSVI 类型信息 */
}
```

8. 节点集合类型 `xmlNodeSet`、指针类型 `xmlNodeSetPtr`

节点集合代表一个由节点组成的变量，节点集合只作为 `xPath` 的查询结果而出现，因此，被定义在 `xpath.h` 中，其定义如下：

```
/* XPath 节点集合定义 */
typedef struct _xmlNodeSet xmlNodeSet;
typedef xmlNodeSet *xmlNodeSetPtr;
struct _xmlNodeSet {
    int nodeNr; /* 节点集合的节点数 */
    int nodeMax; /* 最大可容纳的节点数 */
    xmlNodePtr *nodeTab; /* 节点数组头指针 */
}
```

```
};
```

可以看出，节点集合有三个成员，分别是节点集合的节点数、最大可容纳的节点数、节点数组头指针。对节点集合中各个节点的访问方法如下：

```
xmlNodeSetPtr nodeset = XPath 查询结果;
for (int i = 0; i < nodeset->nodeNr; i++)
{
    nodeset->nodeTab[i];
}
```

20.1.3 libxml 的主要函数说明

许多事物符合 80/20 法则，libxml 中也是 20%的函数提供了 80%的功能。下面列出了 libxml 的主要函数及其用法说明。

1. 全局函数说明

头文件引用	xml2config --cflags
库文件引用	xml2config --libs
主要头文件	parse.h tree.h xpath.h
简单例程	Makefile: INCLUDE=-I./ -I\$HOME/include `xml2config --cflags` LIBRARY=-L./ -L\$HOME/lib `xml2config --libs` *.c 或 *.cpp #include <libxml/parse.h> #include <libxml/tree.h> #include <libxml/xpath.h>

函数功能	在分析 XML 数据时，去除空白字符。如果不去除空白字符，则这些字符也会被当做一个 node 来处理
函数接口	int xmlKeepBlanksDefault(int val)
参数说明	val: 0 或者 1。0 表示去除空白字符，1 表示不去除 返回值: 0 表示设置失败，1 表示设置成功
简单例程	xmlKeepBlanksDefault(0);

2. XML 文件载入和保存函数

函数功能	将 XML 文件从硬盘上载入内存中，并且生成 DOM 树。使用完毕后，需要用 xmlFreeDoc()来释放资源
函数接口	xmlDocPtr xmlParseFile(const char * filename)
参数说明	filename: XML 文件名称。 返回值: 如果载入成功，则返回这个文档的根节点；否则返回 NULL

续表

简单例程	<pre>xmlDocPtr pdoc; pdoc = xmlParseFile("123.xml"); if(pdoc == NULL) { printf("Fail to parse XML file.\n"); }</pre>
------	--

函数功能	将内存中的 DOM 树保存到硬盘上，生成一个带格式的 XML 文件
函数接口	<code>int xmlSaveFormatFileEnc(const char * filename, xmlDocPtr cur, const char * encoding, int format)</code>
参数说明	filename: 需要保存的文件名称 cur: 需要保存的 XML 文档 encoding: 导出文件的编码类型，或者为 NULL format: 是否格式化。0 表示不格式化，1 表示需要格式化。注意：只有当 <code>xmlIndentTreeOutput</code> 设置为 1 或者 <code>xmlKeepBlanksDefault(0)</code> 时，format 设置为 1 才能生效 返回值：写入文件中的字节数量
简单例程	<pre>xmlDocPtr pdoc; pdoc = xmlParseFile("123.xml"); if(pdoc == NULL) { printf("Fail to parse XML file.\n"); } Do_something_with_pdoc(); int filesize; filesize = xmlSaveFormatFileEnc("321.xml", pdoc, "gb2312", 1); if(filesize == -1) { printf("Fail to save XML to file.\n"); }</pre>

3. XML 内存载入和输出函数

函数功能	将一块内存中的 XML 数据生成一个 DOM 树。使用完毕后，需要用 <code>xmlFreeDoc()</code> 来释放资源
函数接口	<code>xmlDocPtr xmlParseMemory(const char * buffer, int size)</code>
参数说明	buffer: 存放 XML 格式数据的内存区 size: 内存中 XML 格式数据的长度 返回值：如果载入成功，则返回这个文档的根节点；否则返回 NULL
简单例程	<pre>char *buffer; int size; xmlDocPtr pdoc; // read_data_to_buffer buffer = size = strlen(buffer); pdoc = xmlParseMemory(buffer, size); if(pdoc == NULL) { printf("Fail to parse XML buffer.\n"); }</pre>

函数功能	将 DOM 树导出到内存中，形成一个 XML 格式的数据
函数接口	<code>void xmlDocDumpFormatMemoryEnc(xmlDocPtr out_doc, xmlChar ** doc_txt_ptr, int * doc_txt_len, const char * txt_encoding, int format)</code>
参数说明	<p><code>out_doc</code>: 需要输出成为一个 buffer 的 XML 文档</p> <p><code>doc_txt_ptr</code>: 输出文档的内存区。由该函数在内部申请，使用完后，必须调用 <code>xmlFree()</code> 函数来释放该内存块</p> <p><code>doc_txt_len</code>: 输出文档内存区的长度</p> <p><code>txt_encoding</code>: 输出文档的编码类型</p> <p><code>format</code>: 是否格式化。0 表示不格式化，1 表示需要格式化。注意：只有当 <code>xmlIndentTreeOutput</code> 设置为 1 或者 <code>xmlKeepBlanksDefault(0)</code> 时，<code>format</code> 设置为 1 才能生效</p>
简单例程	<pre>xmlChar *outbuf; int outlen; xmlDocPtr pdoc; pdoc = ... xmlDocDumpFormatMemoryEnc(pdoc, &outbuf, &outlen, "gb2312", 1); xmlFree(outbuf);</pre>

4. 创建和释放 XML 文档函数

函数功能	在内存中创建一个新的 XML 文档。所创建的文档需要使用 <code>xmlFreeDoc()</code> 来释放资源
函数接口	<code>xmlDocPtr xmlNewDoc(const xmlChar * version)</code>
参数说明	<code>version</code> : XML 标准的版本，目前只能指定为“1.0”
简单例程	<pre>xmlDocPtr pdoc ; pdoc = xmlNewDoc((const xmlChar*)"1.0"); if(pdoc == NULL) { printf("Fail to create new XML doc.\n"); }</pre>

函数功能	释放内存中的 XML 文档
函数接口	<code>void xmlFreeDoc(xmlDocPtr cur)</code>
参数说明	<code>cur</code> : 需要释放的 XML 文档
简单例程	<pre>xmlDocPtr pdoc ; pdoc = xmlNewDoc((const xmlChar*)"1.0"); if(pdoc == NULL) { printf("Fail to create new XML doc.\n"); } xmlFreeDoc(podc);</pre>

5. XML 节点操作函数

函数功能	获得根节点
函数接口	XmlNodePtr xmlDocGetRootElement(xmlDocPtr doc)
参数说明	doc: XML 文档句柄 返回值: XML 文档的根节点, 或者 NULL
使用流程	① 解析好文档的根节点指针, 使用该指针可以遍历 XML 文件 ② xmlDocPtr 的 next 字段, 指向下一个同级 XML 节点 ③ properties 字段为 xmlAttr 类型, 指向该 XML 节点的属性 ④ children 字段为 XmlNodePtr 类型, 指向该 XML 节点的子节点
简单例程	<pre>xmlDocPtr pdoc ; XmlNodePtr root ; pdoc = xmlParseFile("123.xml"); if(pdoc == NULL) { printf("Fail to parse XML File.\n"); return ; } root = xmlDocGetRootElement(pdoc); if(root == NULL) { printf("Fail to get root element\n"); return; }</pre>

函数功能	设置根节点
函数接口	XmlNodePtr xmlDocSetRootElement(xmlDocPtr doc, XmlNodePtr root)
参数说明	doc: XML 文档句柄 root: XML 文档的新的根节点 返回值: 如果该文档原来有根节点, 则返回根节点; 否则返回 NULL
简单例程	<pre>xmlDocPtr pdoc ; XmlNodePtr root; pdoc = xmlNewDoc((const xmlChar*)"1.0"); if(pdoc == NULL) { printf("Fail to create new XML doc.\n"); return; } root = xmlNewDocNode(pdoc, NULL, (const xmlChar*)"root", NULL); if(root == NULL) { printf("Fail to create doc node.\n"); return ; } xmlDocSetRootElement(pdoc, root);</pre>

函数功能	获得节点的内容
函数接口	xmlChar *xmlNodeGetContent(xmlNodePtr cur)
参数说明	cur: 节点的指针 返回值: 节点的文本内容。如果该节点没有文本内容, 则返回 NULL。当返回值不为 NULL 时, 需要用 xmlFree() 函数来释放返回的资源
简单例程	<pre>xmlNodePtr node; xmlChar* content; node = ... content = xmlNodeGetContent(node); xmlFree(content);</pre>

函数功能	设置节点的内容长度
函数接口	void xmlNodeSetContentLen(xmlNodePtr cur, const xmlChar * content, int len)
参数说明	cur: 节点的指针 content: 节点的新文本内容 len: 节点新文本内容的长度
简单例程	<pre>xmlNodePtr node; xmlChar* content; int len; content = (xmlChar*)"1234567890"; len = strlen((char*)content); xmlNodeSetContentLen(node, content, len);</pre>

函数功能	在节点内容的后面添加新的内容
函数接口	void xmlNodeAddContentLen(xmlNodePtr cur, const xmlChar * content, int len)
参数说明	cur: 节点的指针 content: 节点新加的文本内容 len: 节点新加的文本内容的长度
简单例程	<pre>xmlNodePtr node; xmlChar* content; int len; content = (xmlChar*)"1234567890"; len = strlen((char*)content); xmlNodeAddContentLen(node, content, len);</pre>

函数功能	获得节点的属性
函数接口	xmlChar *xmlGetProp(xmlNodePtr node, const xmlChar * name)
参数说明	node: XML 节点的指针 name: 该节点的属性名称 返回值: 该属性的值或者为 NULL。如果不为 NULL, 则需要用 xmlFree() 来释放资源

续表

函数功能	获得节点的属性
简单例程	<pre>XmlNodePtr node; xmlChar* prop; node = ... prop = xmlGetProp(node, (const xmlChar*)"name"); if(prop != NULL) xmlFree(prop);</pre>

函数功能	设置节点的属性（如果该属性已经存在，则替换其值）
函数接口	<pre>xmlAttrPtr xmlSetProp(xmlNodePtr node, const xmlChar * name, const xmlChar * value)</pre>
参数说明	<p>node: 需要设置属性的节点</p> <p>name: 属性的名称</p> <p>value: 属性的值</p> <p>返回值: 该属性节点的指针</p>
简单例程	<pre>XmlNodePtr node; xmlAttrPtr attr; node = ... attr = xmlSetProp(node, (const xmlChar*)"Dept-Name", (const xmlChar*)"ES"); if(attr == NULL) { printf("Fail to set prop of this node.\n"); }</pre>

6. XPath 函数

函数功能	生成 xpath 的上下文关系句柄
函数接口	<pre>xmlXPathContextPtr xmlXPathNewContext(xmlDocPtr doc)</pre>
参数说明	<p>doc: 需要处理的 XML 文档</p> <p>返回值: 该文档的 XPath 上下文关系句柄或者 NULL。该返回句柄由函数内部申请，此函数调用者需要用 xmlXPathFreeContext 来释放</p>
简单例程	<pre>xmlDocPtr pdoc; xmlXPathContextPtr xpathctx; pdoc = ... xpathctx = xmlXPathNewContext(pdoc); if(xpathctx != NULL) xmlXPathFreeContext(xpathctx);</pre>

函数功能	释放 xpath 的上下文关系句柄
函数接口	<pre>void xmlXPathFreeContext(xmlXPathContextPtr ctxt)</pre>
参数说明	ctxt: 需要释放的 xpath 上下文关系句柄
简单例程	参见 xmlXPathNewContext() 的例程

函数功能	执行 xpath 的表达式，返回结果内容节点集合 XPath 表达式的表示方法请参考： http://www.zvon.org/xxl/XPathTutorial/General/examples.html
函数接口	xmlXPathObjectPtr xmlXPathEvalExpression(const xmlChar * str, xmlXPathContextPtr ctxt)
参数说明	str: xpath 表达式 ctxt: xpath 的上下文关系句柄 返回值: 满足表达式的结果集合或者为 NULL。该返回句柄由函数内部申请，此函数调用者需要用 xmlXPathFreeObject()来释放
简单例程	<pre>xmlDocPtr pdoc; xmlXPathContextPtr xpathctx; xmlXPathObjectPtr xpathobj; pdoc = ... xpathctx = xmlXPathNewContext(pdoc); if(xpathctx == NULL) { printf("Fail to create xpath context.\n"); return ; } XPathobj = xmlXPathEvalExpression(BAD_CAST "@value", xpathctx); if(xpathobj == NULL) { printf("Fail to evaluate xpath expression.\n"); xmlXPathFreeContext(xpathctx); return; }</pre> <pre>xmlXPathFreeObject(xpathobj); xmlXPathFreeContext(xpathctx);</pre> <p>结果集说明:</p> <p>xpathobj->nodesetval: 存储结果列表，如果为 NULL，表示无结果</p> <p>xpathObj->nodesetval->nodeNr: 表示结果的数量</p> <p>xpathObj->nodesetval->nodeTab: 表示结果的数组，可以通过下标访问</p> <p>例如:</p> <pre>int number; xmlNodePtr node; if(xpathobj-> nodesetval == NULL) number = 0; else number = xpathObj->nodesetval->nodeNr; for(int i=0;i<number;i++) { node = xpathObj->nodesetval->nodeTab[i]; do_some_work_with_node(); }</pre>

函数功能	释放 xpath 表达式运算结果集
函数接口	void xmlXPathFreeObject(xmlXPathObjectPtr obj)
参数说明	obj: 需要释放的 xpath 表达式运算结果集合
简单例程	参见 xmlXPathEvalExpression() 的例程

7. XML 常见的函数列表

下面是对 XML 常见函数的简要说明，有些函数的具体说明见上面的表格。

```
<libxml/parser.h>
int xmlKeepBlanksDefault (int val)
//设置是否忽略空白节点,比如空格,在分析前必须调用,默认值是 0,最好设置成 1
xmlDocPtr xmlParseFile (const char * filename)
//分析一个 xml 文件,并返回一个文档对象指针

<libxml/tree.h>
//xml 操作的基础结构提及指针类型
//xmlDoc  xmlDocPtr 文档对象的结构体及其指针
//xmlNode  xmlNodePtr 节点对象的结构体及其指针
//xmlAttr  xmlAttrPtr 节点属性的结构体及其指针
//xmlNs  xmlNsPtr 节点命名空间的结构及其指针

//根节点相关函数
xmlNodePtr xmlDocGetRootElement (xmlDocPtr doc) //获取文档根节点
xmlNodePtr xmlDocSetRootElement (xmlDocPtr doc, xmlNodePtr root) //设置文档根节点

//创建子节点相关函数
xmlNodePtr xmlNewNode (xmlNsPtr ns, const xmlChar * name) //创建新节点
xmlNodePtr xmlNewChild (xmlNodePtr parent, xmlNsPtr ns, const xmlChar * name,
const xmlChar * content) //创建新的子节点
xmlNodePtr xmlCopyNode (const xmlNodePtr node, int extended) //复制当前节点

//添加子节点相关函数
xmlNodePtr xmlAddChild (xmlNodePtr parent, xmlNodePtr cur) //给指定节点添加子节点
xmlNodePtr xmlAddNextSibling (xmlNodePtr cur, xmlNodePtr elem)
//添加后一个兄弟节点
xmlNodePtr xmlAddPrevSibling (xmlNodePtr cur, xmlNodePtr elem)
//添加前一个兄弟节点
xmlNodePtr xmlAddSibling (xmlNodePtr cur, xmlNodePtr elem) //添加兄弟节点

//属性相关函数
xmlAttrPtr xmlNewProp (xmlNodePtr node, const xmlChar * name, const xmlChar *
value)
//创建新节点属性
xmlChar * xmlGetProp (xmlNodePtr node, const xmlChar * name) //读取节点属性
xmlAttrPtr xmlSetProp (xmlNodePtr node, const xmlChar * name, const xmlChar * value)
//设置节点属性,作用同尾部同名的字符串函数。只不过针对相应的 xml 节点
xmlChar* xmlStrcat (xmlChar *cur, const xmlChar * add)
const xmlChar *xmlStrchr(const xmlChar * str, xmlChar val)
int xmlStrcmp (const xmlChar * str1, const xmlChar * str2)
int xmlStrlen (const xmlChar * str)
```

```
xmlChar *xmlStrncat (xmlChar * cur, const xmlChar * add, int len)
int xmlStrncmp (const xmlChar * str1, const xmlChar * str2, int len)
const xmlChar *xmlStrstr (const xmlChar * str, const xmlChar * val)
```

20.1.4 XML 常见操作

1. 创建 XML 文档

创建一个 XML 文档非常简单，其流程如下：

- ① 用 `xmlNewDoc` 函数创建一个文档指针 `doc`。
- ② 用 `xmlNewNode` 函数创建一个节点指针 `root_node`。
- ③ 用 `xmlDocSetRootElement` 将 `root_node` 设置为 `doc` 的根节点。
- ④ 给 `root_node` 添加一系列的子节点，并设置子节点的内容和属性。
- ⑤ 用 `xmlSaveFile` 将 XML 文档存入文件。
- ⑥ 用 `xmlFreeDoc` 关闭文档指针，并清除本文档中所有节点动态申请的内存。

有多种方式可以添加子节点，如可以用 `xmlNewTextChild` 直接添加一个文本子节点，也可以先创建新节点，然后用 `xmlAddChild` 将新节点加入上层节点中。

下面是创建 xml 文件的示例。CreateXmlFile.c 源代码如下：

```
#include <stdio.h>
#include <libxml/parser.h>
#include <libxml/tree.h>
int main()
{
    //定义文档和节点指针
    xmlDocPtr doc = xmlNewDoc(BAD_CAST "1.0");
    xmlNodePtr root_node = xmlNewNode(NULL, BAD_CAST "root");
    //设置根节点
    xmlDocSetRootElement(doc, root_node);
    //在根节点中直接创建节点
    xmlNewTextChild(root_node, NULL, BAD_CAST "newNode1", BAD_CAST "newNode1 content");
    xmlNewTextChild(root_node, NULL, BAD_CAST "newNode2", BAD_CAST "newNode2 content");
    xmlNewTextChild(root_node, NULL, BAD_CAST "newNode3", BAD_CAST "newNode3 content");
    //创建一个节点，设置其内容和属性，然后加入根节点
    xmlNodePtr node = xmlNewNode(NULL, BAD_CAST "node2");
    xmlNodePtr content = xmlNewText(BAD_CAST "NODE CONTENT");
    xmlAddChild(root_node, node);
    xmlAddChild(node, content);
    xmlNewProp(node, BAD_CAST "attribute", BAD_CAST "yes");
    //创建一个儿子和孙子节点
    node = xmlNewNode(NULL, BAD_CAST "son");
    xmlAddChild(root_node, node);
    xmlNodePtr grandson = xmlNewNode(NULL, BAD_CAST "grandson");
    xmlAddChild(node, grandson);
    xmlAddChild(grandson, xmlNewText(BAD_CAST "This is a grandson node"));
```



```
//存储 xml 文档
int nRel = xmlSaveFile("CreateXml.xml",doc);
if (nRel != -1)
{
    printf("一个 xml 文档被创建, 写入%d 个字节\n", nRel);
}
//释放文档内节点动态申请的内存
xmlFreeDoc(doc);
return 1;
}
```

编译 `gcc CreateXmlFile.c -o CreateXmlFile -I/usr/local/include/libxml2 -lxml2`。

执行 `./CreateXmlFile`, 会生成一个 XML 文件 `CreatedXml.xml`, 具体代码如下:

```
<?xml version="1.0"?>
<root>
  <newNode1>newNode1 content</newNode1>
  <newNode2>newNode2 content</newNode2>
  <newNode3>newNode3 content</newNode3>
  <node2 attribute="yes">NODE CONTENT</node2>
  <son>
    <grandson>This is a grandson node</grandson>
  </son>
</root>
```

最好使用类似 XMLSPY 这样的工具打开, 因为这些工具可以自动整理 XML 文件的栅格, 否则很有可能是没有任何换行的一个 XML 文件, 可读性较差。

2. 解析 XML 文档

(1) XML 解析流程

解析一个 XML 文档, 从中取出想要的信息, 例如, 节点中包含的文字, 或者某个节点的属性。其流程如下:

- ① 用 `xmlReadFile` 函数读入一个文件, 并返回一个文档指针 `doc`。
- ② 用 `xmlDocGetRootElement` 函数得到根节点 `curNode`。
- ③ 此时 `curNode->xmlChildrenNode` 就是根节点的首个儿子节点, 该儿子节点的兄弟节点可用 `next` 指针进行轮询。
- ④ 轮询所有的子节点, 找到所需的节点, 用 `xmlNodeGetContent` 取出其内容。
- ⑤ 用 `xmlHasProp` 查找含有某个属性的节点, 属性列表指针 `xmlAttrPtr` 将指向该节点的属性列表。
- ⑥ 取出该节点的属性, 用 `xmlGetProp` 取出其属性值。
- ⑦ `xmlFreeDoc` 函数关闭文档指针, 并清除本文档中所有的节点动态申请的内存。

(2) XML 解析举例

ParseXmlFile.c 源代码如下:

```
#include <stdio.h>
#include <libxml/parser.h>
#include <libxml/tree.h>
int main(int argc, char* argv[])
{
    xmlDocPtr doc;           //定义解析文件指针
    xmlNodePtr curNode;      //定义节点指针
    xmlChar *szKey;          //临时字符串变量
    char *szDocName;
    if (argc <= 1)
    {
        printf("Usage: %s docname", argv[0]);
        return(0);
    }
    szDocName = argv[1];
    doc = xmlReadFile(szDocName, "GB2312", XML_PARSE_RECOVER);
    //解析文件
    //检查解析文档是否成功, 如果不成功, libxml 将报错并停止解析
    //一个常见错误是不适当的编码, XML 标准文档除了用 UTF-8 或 UTF-16 外, 还可用其他编码保存
    if (NULL == doc)
    {
        fprintf(stderr, "Document not parsed successfully.");
        return -1;
    }
    //获取根节点
    curNode = xmlDocGetRootElement(doc);
    if (NULL == curNode)
    {
        fprintf(stderr, "empty document");
        xmlFreeDoc(doc);
        return -1;
    }
    //确认根元素名字是否符合
    if (xmlStrcmp(curNode->name, BAD_CAST "root"))
    {
        fprintf(stderr, "document of the wrong type, root node != root");
        xmlFreeDoc(doc);
        return -1;
    }
    curNode = curNode->xmlChildrenNode;
    xmlNodePtr propNodePtr = curNode;
    while(curNode != NULL)
    {
        //取出节点中的内容
        if (!xmlStrcmp(curNode->name, (const xmlChar *) "newNode1"))
        {
            szKey = xmlNodeGetContent(curNode);
            printf("newNode1: %s\n", szKey);
            xmlFree(szKey);
        }
        //查找带有属性 attribute 的节点
```

```

    if (xmlHasProp(curNode,BAD_CAST "attribute"))
    {
        propNodePtr = curNode;
    }
    curNode = curNode->next;
}
//查找属性
xmlAttrPtr attrPtr = propNodePtr->properties;
while (attrPtr != NULL)
{
    if (!xmlStrcmp(attrPtr->name, BAD_CAST "attribute"))
    {
        xmlChar* szAttr = xmlGetProp(propNodePtr,BAD_CAST "attribute");
        printf("get attribute=%s\n", szAttr) ;
        xmlFree(szAttr);
    }
    attrPtr = attrPtr->next;
}
xmlFreeDoc(doc);
return 0;
}

```

编译 `gcc ParseXmlFile.c -o ParseXmlFile -I/usr/local/include/libxml2 -lxml2`。

执行 `./ParseXmlFile CreateXml.xml`，执行结果如下：

```

newNode1: newNode1 content
get attribute=yes

```

3. 修改 XML 文档

有了上面的基础后，修改 XML 文档的内容就很简单了。首先打开一个已经存在的 XML 文档，顺着根节点找到需要添加、删除、修改的地方，调用相应的 XML 函数对节点进行增、删、改操作。

需要注意的是，并没有 `xmlDelNode` 或者 `xmlRemoveNode` 函数，删除节点需使用以下一段代码：

```

if (!xmlStrcmp(curNode->name, BAD_CAST "newNode1"))
{
    xmlNodePtr tempNode;
    tempNode = curNode->next;
    xmlUnlinkNode(curNode);
    xmlFreeNode(curNode);
    curNode = tempNode;
    continue;
}

```

此段代码完成将当前节点从文档中断链(unlink)，这样此 XML 文档就不会再包含这个节点，该节点断链后需使用 `xmlFreeNode` 来释放该节点申请的动态内存空间。

4. 使用 XPath 查找 XML 文档

在 libxml2 中使用 XPath 非常简单，其流程如下：

① 定义一个 XPath 上下文指针 `xmlXPathContextPtr context`, 并且使用 `xmlXPathNewContext` 函数来初始化这个指针。

② 定义一个 XPath 对象指针 `xmlXPathObjectPtr result`, 并使用 `xmlXPathEvalExpression` 函数来计算 XPath 表达式, 将查询结果存入对象指针中。

③ 使用 `result->nodesetval` 得到节点集合指针, 其中包含了所有符合 XPath 查询结果的节点。

④ 使用 `xmlXPathFreeContext` 释放上下文指针。

⑤ 使用 `xmlXPathFreeObject` 释放 XPath 对象指针。

XPath 操作代码示例如下:

```
xmlXPathObjectPtr getNodeSet(xmlDocPtr doc, const xmlChar *szXPath)
{
    xmlXPathContextPtr context;           //XPath 上下文指针
    xmlXPathObjectPtr result;             //XPath 对象指针, 用来存储查询结果
    context = xmlXPathNewContext(doc);     //创建一个 XPath 上下文指针
    if (context == NULL)
    {
        printf("context is NULL\n");
        return NULL;
    }
    result = xmlXPathEvalExpression(szXPath, context);
                                           //查询 XPath 表达式, 得到一个查询结果
    xmlXPathFreeContext(context);         //释放上下文指针
    if (result == NULL)
    {
        printf("xmlXPathEvalExpression return NULL\n");
        return NULL;
    }
    if (xmlXPathNodeSetIsEmpty(result->nodesetval)) //检查查询结果是否为空
    {
        xmlXPathFreeObject(result);
        printf("nodeset is empty\n");
        return NULL;
    }
    return result;
}
```

5. 用 iconv 解决 XML 中字符集问题

libxml2 中默认的内码是 UTF-8, 所有使用 libxml2 进行处理的 xml 文件必须首先显式地或者默认转换为 UTF-8 编码才能被处理。

如果要在 XML 中使用中文, 就必须能够在 UTF-8 和 GB2312 之间进行转换。字符串或文件转码需要使用 iconv 库来进行, libxml2 本身也是使用 iconv 库进行编码转换的。iconv 是一个专门用来进行编码转换的库, 基本上支持目前所有常用的编码转换, 它是 glibc 库的一个部分。

下文提供了一个通用转码函数, 并在此基础上实现了两个转码封装函数, 即从 UTF-8 编码转

换到 GB2312 编码的函数 `u2g`，以及反向转换的函数 `g2u`，其代码如下：

```
#include <iconv.h>
#include <string.h>
//代码转换，从一种编码转为另一种编码
int code_convert(char* from_charset, char* to_charset, char* inbuf,
                 int inlen, char* outbuf, int outlen)
{
    iconv_t cd;
    char** pin = &inbuf;
    char** pout = &outbuf;
    cd = iconv_open(to_charset, from_charset);
    if(cd == 0)
        return -1;
    memset(outbuf, 0, outlen);
    if(iconv(cd, (const char**)pin, (unsigned int *)&inlen, pout, (unsigned int *)&outlen)
        == -1)
        return -1;
    iconv_close(cd);
    return 0;
}
//UNICODE 码转为 GB2312 码
//成功则返回一个动态分配的 char*变量，在使用完毕后需要手动释放（free），失败则返回 NULL
char* u2g(char *inbuf)
{
    int nOutLen = 2 * strlen(inbuf) - 1;
    char* szOut = (char*)malloc(nOutLen);
    if (-1 == code_convert("utf-8", "gb2312", inbuf, strlen(inbuf), szOut, nOutLen))
    {
        free(szOut);
        szOut = NULL;
    }
    return szOut;
}
//GB2312 码转为 UNICODE 码
//成功则返回一个动态分配的 char*变量，在使用完毕后需要手动释放（free），失败则返回 NULL
char* g2u(char *inbuf)
{
    int nOutLen = 2 * strlen(inbuf) - 1;
    char* szOut = (char*)malloc(nOutLen);
    if (-1 == code_convert("gb2312", "utf-8", inbuf, strlen(inbuf), szOut, nOutLen))
    {
        free(szOut);
        szOut = NULL;
    }
    return szOut;
}
```

下面以 UTF-8 到 GB2312 转码流程说明上文中转码函数的使用，使用流程如下：

- ① 得到一个 UTF-8 的字符串 `szSrc`。
- ② 定义一个 `char *` 的字符指针 `szDes`，并不需要给它动态申请内存。

- ③ 调用 `szDes = u2g(szSrc)`，这样 `szDes` 就指向转换后 GB2312 编码的字符串。
- ④ 使用完这个字符串后使用 `free(szDes)` 来释放内存。

上文的转码函数主要用于一片内存空间内容的转码，也可以在此基础上实现文件的转码。如果只是对文件进行转码，选择通过 `system` 系统调用方式实现起来更简单。系统调用实现文件转码的方法如下：

```
system("iconv -f 源格式 -t 目标格式 源文件 >目标文件")
system("iconv -f GB18030 -t UTF-8 test_gb.txt > test_utf.txt")
```

20.2 libxml 高级编程进阶

20.2.1 理解 DOM 树

理解 DOM 树后能更好地理解 libxml 函数的操作原理。为了大家更好地理解 DOM 树在内存中的构造，本书特地写了一个典型的 XML 文件 (`dom.xml`)，并且画出了其 DOM 树内存构造图，其后提供了一个不用 XML 库查找 XML 节点和属性的通用程序 (`dom_xml.c`)，此程序具有较好的实用性。

1. dom.xml 文件


`dom.xml` 文件内容如下：

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<Format FmtName="FMT_DXPT_DX0001_IN"
    FmtDesc="dxpt fmt"
    Fmttype="string">
    <meta>
        <hello>first</hello>
        <good>second</good>
        <gold>third</gold>
    </meta>
    <metal>
        <what>why</what>
    </metal>
</Format>
```

2. dom.xml 形成的 DOM 树

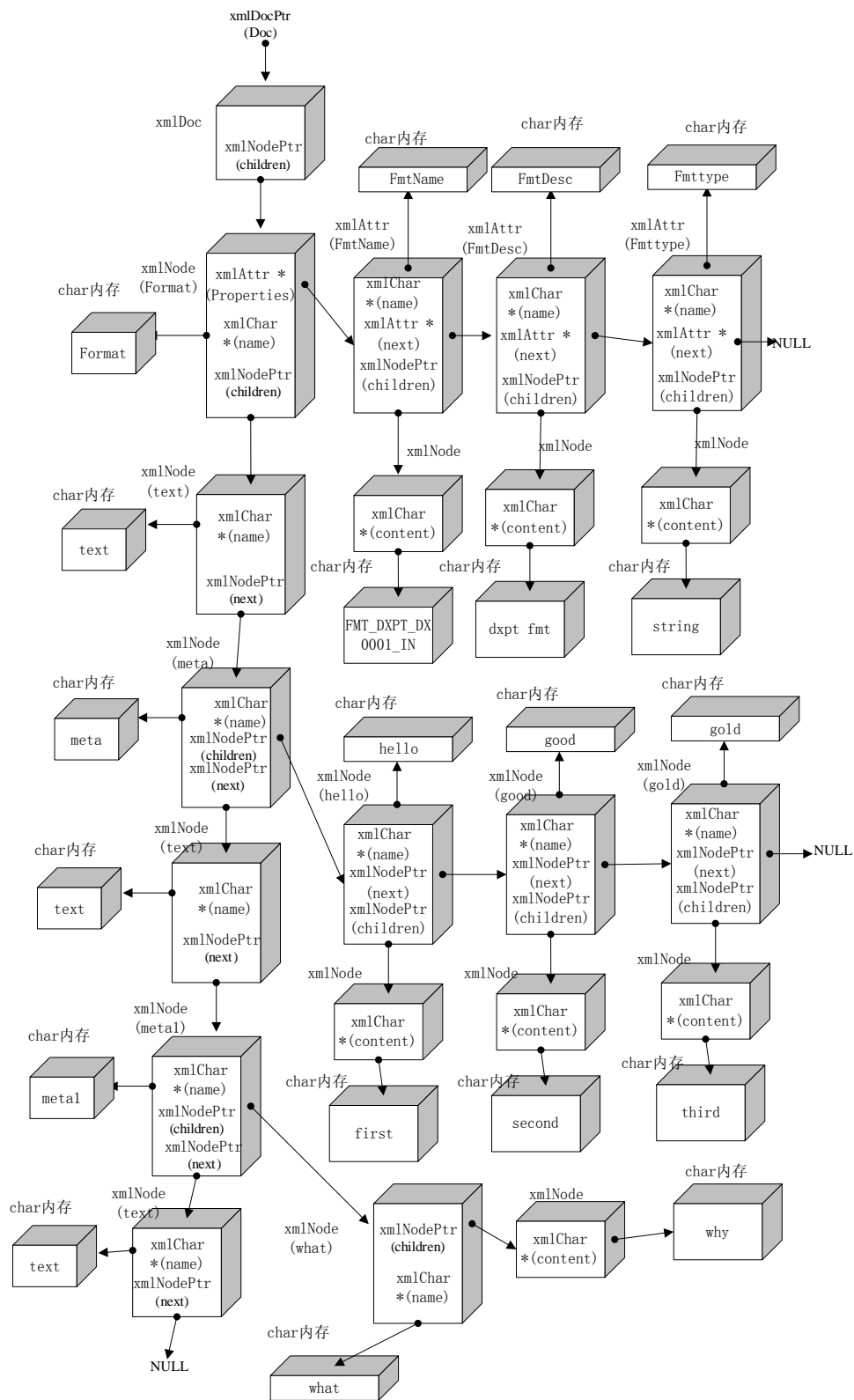
图 20-1 形象地说明了 `dom.xml` 在内存中是如何形成 DOM 树的，其中：

- > 表示指针，指针指向一片内存。

 表示申请的一片内存空间。

`xmlNode(Format)` 表示这片内存空间类型为 `xmlNode`，元素名称为 `Format`。

`xmlNodePtr` 为 `xmlNode *` 的重定义，即 `xmlNode` 的指针重定义。



3. 得到 XML 节点和属性的通用程序

dom_xml.c 源代码如下:

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>
#include <assert.h>
#include <libxml/xmlmemory.h>
#include <libxml/parser.h>
#include <libxml/xpath.h>
/*项目中数字最好用宏代替*/
struct XML_attr {
    char attr_name[30+1] ;
    char attr_value[100+1] ;
} ;
struct XML_node{
    char node_name[30+1] ;
    char node_value[100+1] ;
} ;
/*-----
 * Function Name : get_xml_attr
 * Description   : 得到节点的所有属性
 * Input         : node_ptr      ——当前节点
 *               : xml_attr      ——存储属性名称和属性值
 *               : maxnum        ——最大属性个数
 *               : -1 -- Failure
 *-----*/
int get_xml_attr( xmlDocPtr node_ptr, struct XML_attr xml_attr[], int maxnum)
{
    xmlAttr      *attr;
    int i = 0;
    attr = node_ptr->properties;
    while( attr != NULL ) {
        if ( attr->children == NULL ) {
            attr = attr->next;
            continue;
        }
        strncpy( xml_attr[i].attr_name, ( char * )attr->name, 30 );
        strncpy( xml_attr[i].attr_value, ( char * )attr->children->content,
100 );

        attr = attr->next;
        i++ ;
        if ( i > maxnum)
        {
            printf("xml attr over limit num\n");
            return -1 ;
        }
    }
    return 0 ;
}
/*-----
 * Function Name : get_xml_node
```



```

* Description   : 得到节点的所有属性
* Input        : node_ptr   ——当前节点
*              : xml_node   ——存储元素名称和值
*              : maxnum     ——最大元素个数
*Output       : xml_node   ——存储元素名称和值
* Return      : 0  -- Success
*             -1  -- Failure
*-----*/
int get_xml_node( xmlNodePtr node_ptr, struct XML_node xml_node[], int maxnum)
{
    xmlNodePtr    cur;
    int i = 0;
    cur = node_ptr->children;
    while( cur != NULL ) {
        if ( cur->children == NULL ) {
            cur = cur->next;
            continue;
        }
        strncpy( xml_node[i].node_name, ( char * )cur->name, 30 );
        strncpy( xml_node[i].node_value, ( char * )cur->children->content, 100 );
        cur = cur->next;
        i++;
        if ( i > maxnum )
        {
            printf("xml node over limit num\n");
            return -1 ;
        }
    }
    return 0 ;
}
/*
*   Function Name: get_xml_node_ptr
*   Description   : 获取子节点指针
*   Input        : node_ptr ——节点指针
*               : node_name ——节点名
*   Output       : node_ptr ——子节点指针
*   Return      : 0  -- Success
*               -1  -- Failure
*/
int get_xml_node_ptr( xmlNodePtr node_ptr , char *node_name )
{
    xmlNodePtr    cur;
    cur = node_ptr;
    while( cur != NULL ) {
        if( ( xmlStrcmp( cur->name, node_name ) ) == 0 )
        {
            break;
        }
        else
            cur = cur->next;
    }
    if( cur == NULL ) {
        return -1;
    }
    node_ptr = cur;
}

```

```

    return 0 ;
}
int main()
{
    char value[128];
    int ret;
    int i ;
    xmlDocPtr doc;
    xmlNodePtr cur;
    xmlNodePtr attr;
    xmlNodePtr node;
    struct XML_attr fmt_attr[3] ;
    struct XML_node meta_node[3] ;
    memset(fmt_attr, 0x00, sizeof(fmt_attr)) ;
    memset(meta_node, 0x00, sizeof(meta_node)) ;
    doc = xmlParseFile("dom.xml");
    if (doc == NULL ) {
        fprintf(stderr,"Document not parsed successfully. \n");
        return (-1);
    }
    cur = xmlDocGetRootElement(doc);
    if (cur == NULL) {
        fprintf(stderr,"empty document\n");
        xmlFreeDoc(doc);
        return (-1);
    }
    attr = cur ;
    ret=get_xml_attr(attr, fmt_attr, 3) ;
    assert(!ret);
    for(i=0; i<3; i++)
    {
        printf("attr_name[%d]=%s, attr_value[%d]=%s\n",i,fmt_attr[i].attr_name,
i,fmt_attr[i].attr_value);
    }
    node=cur->children ;
    ret = get_xml_node_ptr( node , "meta" ) ;
    assert(!ret);
    node=cur->children->next ;
    ret =get_xml_node(node, meta_node, 3) ;
    assert(!ret);
    for(i=0; i<3; i++)
    {
        printf("node_name[%d]=%s, node_value[%d]=%s\n",i,meta_node[i].node_name,
i,meta_node[i].node_value);
    }
    xmlFreeDoc(doc);
    return 0;
}

```

编译 gcc dom_xml.c -o dom_xml -I/usr/local/include/libxml2 -lxml2。

执行 ./dom_xml, 执行结果如下:

```

attr_name[0]=FmtName, attr_value[0]=FMT_DXPT_DX0001_IN
attr_name[1]=FmtDesc, attr_value[1]=dxpt fmt
attr_name[2]=Fmttype, attr_value[2]=string

```

```
node_name[0]=hello, node_value[0]=first
node_name[1]=good, node_value[1]=second
node_name[2]=gold, node_value[2]=third
```

20.2.2 libxml 编程实例练习

下面提供了五个编程实例，这五个编程实例有助于大家更好地理解和掌握 libxml 编程。

1. 代码中使用的 XML 文件

story.xml 内容如下：

```
<?xml version="1.0"?>
<story>
  <storyinfo>
    <author>John Fleck</author>
    <datewritten>June 2, 2002</datewritten>
    <keyword>example keyword</keyword>
  </storyinfo>
  <body>
    <headline>This is the headline</headline>
    <para>This is the body text.</para>
  </body>
</story>
```

2. 查找 keyword 标记的值

xmlkey.c 源代码如下：

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <libxml/xmlmemory.h>
#include <libxml/parser.h>
void parseStory (xmlDocPtr doc, xmlNodePtr cur) {
    xmlChar *key;
    cur = cur->xmlChildrenNode;
    while (cur != NULL) {
        if (!xmlStrcmp(cur->name, (const xmlChar *)"keyword")) {
            key = xmlNodeListGetString(doc, cur->xmlChildrenNode, 1);
            printf("keyword: %s\n", key);
            xmlFree(key);
        }
        cur = cur->next;
    }
    return;
}
static void parseDoc(char *docname) {
    xmlDocPtr doc;
    xmlNodePtr cur;
    doc = xmlParseFile(docname);
    if (doc == NULL) {
        fprintf(stderr, "Document not parsed successfully. \n");
        return;
    }
}
```

```

cur = xmlDocGetRootElement(doc);
if (cur == NULL) {
    fprintf(stderr, "empty document\n");
    xmlFreeDoc(doc);
    return;
}
if (xmlStrcmp(cur->name, (const xmlChar *) "story")) {
    fprintf(stderr, "document of the wrong type, root node != story");
    xmlFreeDoc(doc);
    return;
}
cur = cur->xmlChildrenNode;
while (cur != NULL) {
    if ((!xmlStrcmp(cur->name, (const xmlChar *) "storyinfo"))){
        parseStory (doc, cur);
    }
    cur = cur->next;
}
xmlFreeDoc(doc);
return;
}
int main(int argc, char **argv) {
    char *docname;
    if (argc <= 1) {
        printf("Usage: %s docname\n", argv[0]);
        return(0);
    }
    docname = argv[1];
    parseDoc (docname);
    return (1);
}

```

编译 `gcc xmlkey.c -o xmlkey -I/usr/local/include/libxml2 -lxml2`。

执行 `./xmlkey story.xml`, 执行结果如下:

keyword: example keyword

3. 利用 XPath 寻找标记

`xmlpath.c` 源代码如下:

```

#include <libxml/parser.h>
#include <libxml/xpath.h>
xmlDocPtr getdoc (char *docname) {
    xmlDocPtr doc;
    doc = xmlParseFile(docname);
    if (doc == NULL ) {
        fprintf(stderr, "Document not parsed successfully. \n");
        return NULL;
    }
    return doc;
}
xmlXPathObjectPtr getnodeset (xmlDocPtr doc, xmlChar *xpath){
    xmlXPathContextPtr context;
    xmlXPathObjectPtr result;

```

```

context = xmlXPathNewContext(doc);
if (context == NULL) {
    printf("Error in xmlXPathNewContext\n");
    return NULL;
}
result = xmlXPathEvalExpression(xpath, context);
xmlXPathFreeContext(context);
if (result == NULL) {
    printf("Error in xmlXPathEvalExpression\n");
    return NULL;
}
if(xmlXPathNodeSetIsEmpty(result->nodesetval)){
    xmlXPathFreeObject(result);
    printf("No result\n");
    return NULL;
}
return result;
}
int main(int argc, char **argv) {
    char *docname;
    xmlDocPtr doc;
    xmlChar *xpath = (xmlChar*) "//keyword"; /*双斜线表示递归全文*/
    xmlNodeSetPtr nodeset;
    xmlXPathObjectPtr result;
    int i;
    xmlChar *keyword;
    if (argc <= 1) {
        printf("Usage: %s docname\n", argv[0]);
        return(0);
    }
    docname = argv[1];
    doc = getdoc(docname);
    result = getnodeset (doc, xpath);
    if (result) {
        nodeset = result->nodesetval;
        for (i=0; i < nodeset->nodeNr; i++) {
            keyword = xmlNodeListGetString(doc,nodeset->nodeTab[i]->xmlChildrenNode, 1);
            printf("keyword: %s\n", keyword);
            xmlFree(keyword);
        }
        xmlXPathFreeObject (result);
    }
    xmlFreeDoc(doc);
    xmlCleanupParser();
    return (1);
}

```

编译 `gcc xmlpath.c -o xmlpath -I/usr/local/include/libxml2 -lxml2`。

执行 `./xmlpath story.xml`，执行结果如下：

```
keyword: example keyword
```

4. 增加节点

`xmladdnode.c` 源代码如下：

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <libxml/xmlmemory.h>
#include <libxml/parser.h>

void parseStory (xmlDocPtr doc, xmlNodePtr cur, char *keyword) {
    xmlNewTextChild (cur, NULL, "keyword", keyword);
    return;
}

xmlDocPtr parseDoc(char *docname, char *keyword) {
    xmlDocPtr doc;
    xmlNodePtr cur;
    doc = xmlParseFile(docname);
    if (doc == NULL ) {
        fprintf(stderr, "Document not parsed successfully. \n");
        return (NULL);
    }
    cur = xmlDocGetRootElement(doc);
    if (cur == NULL) {
        fprintf(stderr, "empty document\n");
        xmlFreeDoc(doc);
        return (NULL);
    }
    if (xmlStrcmp(cur->name, (const xmlChar *) "story")) {
        fprintf(stderr, "document of the wrong type, root node != story");
        xmlFreeDoc(doc);
        return (NULL);
    }
    cur = cur->xmlChildrenNode;
    while (cur != NULL) {
        if ((!xmlStrcmp(cur->name, (const xmlChar *) "storyinfo"))){
            parseStory (doc, cur, keyword);
        }
        cur = cur->next;
    }
    return(doc);
}

int main(int argc, char **argv) {
    char *docname;
    char *keyword;
    xmlDocPtr doc;
    if (argc <= 2) {
        printf("Usage: %s docname, keyword\n", argv[0]);
        return(0);
    }
    docname = argv[1];
    keyword = argv[2];
    doc = parseDoc (docname, keyword);
    if (doc != NULL) {
        xmlSaveFormatFile (docname, doc, 0);
        xmlFreeDoc(doc);
    }
    return (0);
}

```

编译 gcc xmladdnode.c -o xmladdnode -I/usr/local/include/libxml2 -lxml2。

执行 `./xmladdnode story.xml XXX`, `story.xml` 文件中 `keyword` 的值如下:

```
<keyword>XXX</keyword>
```

5. 增加节点和属性

`xmladdattr.c` 源代码如下:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <libxml/xmlmemory.h>
#include <libxml/parser.h>
xmlDocPtr parseDoc(char *docname, char *uri) {
    xmlDocPtr doc;
    xmlNodePtr cur;
    xmlNodePtr newnode;
    xmlAttrPtr newattr;
    doc = xmlParseFile(docname);
    if (doc == NULL) {
        fprintf(stderr, "Document not parsed successfully. \n");
        return (NULL);
    }
    cur = xmlDocGetRootElement(doc);
    if (cur == NULL) {
        fprintf(stderr, "empty document\n");
        xmlFreeDoc(doc);
        return (NULL);
    }
    if (xmlStrcmp(cur->name, (const xmlChar *) "story")) {
        fprintf(stderr, "document of the wrong type, root node != story");
        xmlFreeDoc(doc);
        return (NULL);
    }
    newnode = xmlNewTextChild(cur, NULL, "reference", NULL);
    newattr = xmlNewProp (newnode, "uri", uri);
    return(doc);
}
int main(int argc, char **argv) {
    char *docname;
    char *uri;
    xmlDocPtr doc;
    if (argc <= 2) {
        printf("Usage: %s docname, uri\n", argv[0]);
        return(0);
    }
    docname = argv[1];
    uri = argv[2];
    doc = parseDoc (docname, uri);
    if (doc != NULL) {
        xmlSaveFormatFile (docname, doc, 1);
        xmlFreeDoc(doc);
    }
    return (0);
}
```

编译 `gcc xmladdattr.c -o xmladdattr -I/usr/local/include/libxml2 -lxml2`。

执行 `./xmladdattr story.xml www.163.com`, 结果 `story.xml` 文件内容如下:

```
<?xml version="1.0"?>
<story>
  <storyinfo>
    <author>John Fleck</author>
    <datewritten>June 2, 2002</datewritten>
    <keyword>example keyword</keyword>
  </storyinfo>
  <body>
    <headline>This is the headline</headline>
    <para>This is the body text.</para>
  </body>
  <reference uri="www.163.com"/>
</story>
```

6. 查找属性

`xmlfindattr.c` 源代码如下:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <libxml/xmlmemory.h>
#include <libxml/parser.h>

void getReference (xmlDocPtr doc, xmlNodePtr cur) {
    xmlChar *uri;
    cur = cur->xmlChildrenNode;
    while (cur != NULL) {
        if ((!xmlStrcmp(cur->name, (const xmlChar *)"reference"))) {
            uri = xmlGetProp(cur, "uri");
            printf("uri: %s\n", uri);
            xmlFree(uri);
        }
        cur = cur->next;
    }
    return;
}

void parseDoc(char *docname) {
    xmlDocPtr doc;
    xmlNodePtr cur;
    doc = xmlParseFile(docname);
    if (doc == NULL) {
        fprintf(stderr, "Document not parsed successfully. \n");
        return;
    }
    cur = xmlDocGetRootElement(doc);
    if (cur == NULL) {
        fprintf(stderr, "empty document\n");
        xmlFreeDoc(doc);
        return;
    }
    if (xmlStrcmp(cur->name, (const xmlChar *)"story")) {
```



```

        fprintf(stderr, "document of the wrong type, root node != story");
        xmlFreeDoc(doc);
        return;
    }
    getReference (doc, cur);
    xmlFreeDoc(doc);
    return;
}

int main(int argc, char **argv) {
    char *docname;
    if (argc <= 1) {
        printf("Usage: %s docname\n", argv[0]);
        return(0);
    }
    docname = argv[1];
    parseDoc (docname);
    return (0);
}

```

编译 `gcc xmlfindattr.c -o xmlfindattr -I/usr/local/include/libxml2 -lxml2`。

执行 `./xmlfindattr story.xml`，执行结果如下：

```
uri: www.163.com
```

20.2.3 支付宝银行端接口 XML 项目案例

XML 广泛应用在行业软件中，下面列出的是笔者做银行前置项目时，银行对支付宝接口的实用 XPath 库，此库短小精悍。利用此库几乎可以实现 XML 的所有操作，而且使用简单，此库可以广泛应用在 XML 项目环境中。

1. 实用 XPath 库文件

xpathlib.c 源文件代码如下：

```

#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>
#include <libxml/xmlmemory.h>
#include <libxml/parser.h>
#include <libxml/xpath.h>
/*-----
 * Function Name : getNodePtrSet
 * Description   : 根据 xpath 取 xml 节点集合
 * Input        : xpath    -- xpath 路径
 *               : doc      -- xmlDoc
 * Output       :
 * Return        : 成功: 返回 xmlNodeSetPtr.
 *               : 失败: 返回 NULL
 *-----*/
xmlNodeSetPtr getNodeSet( char *xpath, xmlDocPtr doc )
{

```

```

xmlXPathContextPtr ctx = NULL;
xmlXPathObjectPtr xnode = NULL;
ctx = xmlXPathNewContext(doc);
xnode = xmlXPathEvalExpression( (const xmlChar*)xpath, ctx );
xmlXPathFreeContext(ctx);
return xnode->nodesetval;
}
/*-----
* Function Name : getNodePtr
* Description   : 根据 xpath 取 xml 节点
* Input        : xpath  -- xpath 路径
*               : doc    -- xmlDoc
* Output       :
* Return       : 成功: 返回 xmlNodePtr
*               失败: 返回 NULL
*-----*/
xmlNodePtr getNodePtr( char *xpath, xmlDocPtr doc )
{
    xmlXPathContextPtr ctx = NULL;
    xmlXPathObjectPtr xnode = NULL;
    ctx = xmlXPathNewContext(doc);
    xnode = xmlXPathEvalExpression( (const xmlChar*)xpath, ctx );
    xmlXPathFreeContext(ctx);
    if ( xnode->nodesetval == NULL )
    {
        return NULL;
    }
    if ( xnode->nodesetval->nodeNr == 0 )
    {
        return NULL;
    }
    return xnode->nodesetval->nodeTab[0];
}
/*-----
* Function Name : getNodeName
* Description   : 取 XML 节点的节点名称
* Input        : xpath  -- xpath 路径
*               : doc    -- xmlDoc
* Output       : value   -- 返回节点名
* Return       : 0  -- 成功
*               -1  -- 失败
*-----*/
int getNodeName( char *value, char *xpath, xmlDocPtr doc )
{
    xmlNodePtr node = getNodePtr( xpath, doc );
    if ( node == NULL )
        return -1;
    sprintf( value, "%s", node->name );
    return 0 ;
}
/*-----
* Function Name : getNodeValue
* Description   : 取 XML 节点的值
* Input        : xpath  -- xpath 路径
*               : doc    -- xmlDoc

```

```

* Output      : value  -- 返回变量值
* Return      : 0  -- Success
*            -1  -- Failure
*-----*/
int getNodeValue( char *value, char *xpath, xmlDocPtr doc )
{
    xmlNodePtr node = getNodePtr( xpath, doc );
    if ( node == NULL )
        return -1;
    if ( node->children == NULL )
        return -1;
    sprintf( value, "%s", node->children->content );
    return 0 ;
}
/*-----*/
* Function Name : setNodeName
* Description   : 设置 XML 节点的值
* Input        : value  -- 变量值
*              : xpath  -- xpath 路径
*              : doc    -- xmlDoc
* Return       : 0  -- 成功
*            -1  -- 失败
*-----*/
int setNodeValue( char *value, char *xpath, xmlDocPtr doc )
{
    xmlNodePtr node = getNodePtr( xpath, doc );
    if ( node == NULL )
    {
        printf( "Node not found [%s]\n",xpath) ;
        return -1;
    }
    if ( node->children != NULL )
    {
        xmlUnlinkNode(node->children);
        xmlFreeNode(node->children);
    }
    if ( xmlAddChild( node, xmlNewText( BAD_CAST value ) ) == NULL )
    {
        printf( "xmlAddChild fail\n" );
        return -1;
    }
    return 0 ;
}

```

2. XPath 库测试程序

xpath.c 源代码如下:

```

#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>
#include <assert.h>
#include <libxml/xmlmemory.h>

```

```

#include <libxml/parser.h>
#include <libxml/xpath.h>
int main()
{
    char xmlbuff[1024];
    char value[128];
    int ret;
    xmlNodePtr node1, node2;
    sprintf ( xmlbuff, "%s", "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n"
        "<Cartoon>\n"
        "    <Message id=\"msg001\"> \n"
        "        <BQReq id=\"BQReq001\"> \n"
        "            <version>1.0.1</version> \n"
        "            <date>20080722 09:16:11</date> \n"
        "            <instId>ALIPAY</instId> \n"
        "            <certId>ALIPAY2007072400</certId> \n"
        "            <signNo>162215E52CFE8CC8E8F8F4E2B8A80438</signNo> \n"
        "        </BQReq> \n"
        "    </Message> \n"
        "    <Message id=\"msg002\"> \n"
        "        <BQReq id=\"BQReq002\"> \n"
        "            <version>1.0.1</version> \n"
        "            <date>20080722 09:16:11</date> \n"
        "            <instId>9999</instId> \n"
        "            <certId>8888</certId> \n"
        "            <signNo>162215E52CFE8CC8E8F8F4E2B8A80438</signNo> \n"
        "        </BQReq> \n"
        "    </Message> \n"
        "</Cartoon> \n" );
    xmlDocPtr doc = xmlParseMemory( xmlbuff, strlen(xmlbuff) );
    /*取节点名称*/
    ret = getNodeName( value, "/Cartoon/Message/*", doc );
    assert(!ret);
    printf("value 1=%s\n", value);
    /*取节点属性的值*/
    ret = getNodeValue( value, "/Cartoon/Message/*/@id", doc );
    assert(!ret);
    /*取节点元素的值*/
    ret = getNodeValue( value, "/Cartoon/Message/*/@date", doc );
    assert(!ret);
    printf("value 2=%s\n", value);
    /*重置节点属性的值*/
    ret = setNodeValue( "test002", "/Cartoon/Message[position()=2]/*/@id", doc );
    assert(!ret);
    /*增加节点属性*/
    node1 = getNodePtr("/Cartoon/Message[position()=2]/BQReq",doc);
    if ( node1 != NULL )
    {
        node2 = node1->children;
        xmlNewProp( node1, BAD_CAST "prop", BAD_CAST"add new prop" );
    }
    xmlDocDump( stdout, doc );
    return 0;
}

```

编译 gcc xpath.c xpathlib.c -o xpath -I/usr/local/include/libxml2 -lxml2。

执行 `./xpath`, 执行结果如下:

```
value 1=BQReq
value 2=20080722 09:16:11
<?xml version="1.0" encoding="UTF-8"?>
<Cartoon>
  <Message id="msg001">
    <BQReq id="BQReq001">
      <version>1.0.1</version>
      <date>20080722 09:16:11</date>
      <instId>ALIPAY</instId>
      <certId>ALIPAY2007072400</certId>
      <signNo>162215E52CFE8CC8E8F8F4E2B8A80438</signNo>
    </BQReq>
  </Message>
  <Message id="msg002">
    <BQReq id="test002" prop="add new prop">
      <version>1.0.1</version>
      <date>20080722 09:16:11</date>
      <instId>9999</instId>
      <certId>8888</certId>
      <signNo>162215E52CFE8CC8E8F8F4E2B8A80438</signNo>
    </BQReq>
  </Message>
</Cartoon>
```

附录

1. ASCII 码表

八进制数	十六进制数	十进制数	字符	八进制数	十六进制数	十进制数	字符
00	00	0	nul	100	40	64	@
01	01	1	soh	101	41	65	A
02	02	2	stx	102	42	66	B
03	03	3	etx	103	43	67	C
04	04	4	eot	104	44	68	D
05	05	5	enq	105	45	69	E
06	06	6	ack	106	46	70	F
07	07	7	bel	107	47	71	G
10	08	8	bs	110	48	72	H
11	09	9	ht	111	49	73	I
12	0a	10	nl	112	4a	74	J
13	0b	11	vt	113	4b	75	K
14	0c	12	ff	114	4c	76	L
15	0d	13	er	115	4d	77	M
16	0e	14	so	116	4e	78	N
17	0f	15	si	117	4f	79	O
20	10	16	dle	120	50	80	P
21	11	17	dc1	121	51	81	Q
22	12	18	dc2	122	52	82	R
23	13	19	dc3	123	53	83	S
24	14	20	dc4	124	54	84	T
25	15	21	nak	125	55	85	U
26	16	22	syn	126	56	86	V
27	17	23	etb	127	57	87	W
30	18	24	can	130	58	88	X
31	19	25	em	131	59	89	Y

续表

八进制数	十六进制数	十进制数	字符	八进制数	十六进制数	十进制数	字符
32	1a	26	sub	132	5a	90	z
33	1b	27	esc	133	5b	91	[
34	1c	28	fs	134	5c	92	\
35	1d	29	gs	135	5d	93]
36	1e	30	re	136	5e	94	^
37	1f	31	us	137	5f	95	_
40	20	32	sp	140	60	96	'
41	21	33	!	141	61	97	a
42	22	34	"	142	62	98	b
43	23	35	#	143	63	99	c
44	24	36	\$	144	64	100	d
45	25	37	%	145	65	101	e
46	26	38	&	146	66	102	f
47	27	39	`	147	67	103	g
50	28	40	(150	68	104	h
51	29	41)	151	69	105	i
52	2a	42	*	152	6a	106	j
53	2b	43	+	153	6b	107	k
54	2c	44	,	154	6c	108	l
55	2d	45	-	155	6d	109	m
56	2e	46	.	156	6e	110	n
57	2f	47	/	157	6f	111	o
60	30	48	0	160	70	112	p
61	31	49	1	161	71	113	q
62	32	50	2	162	72	114	r
63	33	51	3	163	73	115	s
64	34	52	4	164	74	116	t
65	35	53	5	165	75	117	u
66	36	54	6	166	76	118	v
67	37	55	7	167	77	119	w
70	38	56	8	170	78	120	x
71	39	57	9	171	79	121	y
72	3a	58	:	172	7a	122	z
73	3b	59	;	173	7b	123	{
74	3c	60	<	174	7c	124	
75	3d	61	=	175	7d	125	}
76	3e	62	>	176	7e	126	~
77	3f	63	?	177	7f	127	del

2. ASCII 码控制或特殊字符说明

Bin	Dec	Hex	缩写/字符	解 释
00000000	0	00	NUL(null)	空字符
00000001	1	01	SOH(start of headling)	标题开始
00000010	2	02	STX (start of text)	正文开始
00000011	3	03	ETX (end of text)	正文结束
00000100	4	04	EOT (end of transmission)	传输结束
00000101	5	05	ENQ (enquiry)	请求
00000110	6	06	ACK (acknowledge)	收到通知
00000111	7	07	BEL (bell)	响铃
00001000	8	08	BS (backspace)	退格
00001001	9	09	HT (horizontal tab)	水平制表符
00001010	10	0A	LF (NL line feed, new line)	换行键
00001011	11	0B	VT (vertical tab)	垂直制表符
00001100	12	0C	FF (NP form feed, new page)	换页键
00001101	13	0D	CR (carriage return)	回车键
00001110	14	0E	SO (shift out)	不用切换
00001111	15	0F	SI (shift in)	启用切换
00010000	16	10	DLE (data link escape)	数据链路转义
00010001	17	11	DC1 (device control 1)	设备控制 1
00010010	18	12	DC2 (device control 2)	设备控制 2
00010011	19	13	DC3 (device control 3)	设备控制 3
00010100	20	14	DC4 (device control 4)	设备控制 4
00010101	21	15	NAK (negative acknowledge)	拒绝接收
00010110	22	16	SYN (synchronous idle)	同步空闲
00010111	23	17	ETB (end of trans. block)	传输块结束
00011000	24	18	CAN (cancel)	取消
00011001	25	19	EM (end of medium)	介质中断
00011010	26	1A	SUB (substitute)	替补
00011011	27	1B	ESC (escape)	溢出
00011100	28	1C	FS (file separator)	文件分割符
00011101	29	1D	GS (group separator)	分组符
00011110	30	1E	RS (record separator)	记录分离符
00011111	31	1F	US (unit separator)	单元分隔符
00100000	32	20	(space)	空格

参考文献

- [1] 谭浩强. C 程序设计. 第 3 版. 北京: 清华大学出版社, 2009
- [2] 宋劲杉. Linux C 编程一站式学习. 北京: 电子工业出版社, 2009
- [3] 吴岳等. Linux C 程序设计大全. 北京: 清华大学出版社, 2009
- [4] [美]林登 (LinDen, P.V.D) 著. C 专家编程. 徐波译. 北京: 人民邮电出版社, 2008
- [5] 孙琼. 嵌入式 Linux 应用程序开发详解. 北京: 人民邮电出版社, 2007
- [6] (美)史蒂文斯, (美)拉戈 著. UNIX 环境高级编程. 尤晋元, 张亚英, 戚正伟 译. 北京: 人民邮电出版社, 2006
- [7] (美) David Tansley 著. Linux 与 UNIX Shell 编程指南. 张春萌等译. 北京: 机械工业出版社, 2000
- [8] 刘冰, 赵廷涛, 邵文豪, 孙兴义. Linux C 程序基础与实例讲解. 北京: 清华大学出版社, 2009
- [9] 赵炯. Linux 内核完全注释. 北京: 机械工业出版社, 2004
- [10] 陈正冲. C 语言深度解剖. 北京: 北京航空航天大学出版社, 2010
- [11] 孙琼. 嵌入式 Linux 应用程序开发详解. 北京: 人民邮电出版社, 2006
- [12] 徐千洋. Linux C 函数库详解词典. 北京: 机械工业出版社, 2008
- [13] http://www.blogjava.net/wxb_nudt/archive/2007/11/28/161340.html
- [14] http://blogold.chinaunix.net/u1/56834/showart_441723.html
- [15] <http://hi.baidu.com/superdbbs/blog/item/13b2d72a9659bf24d42af132.html>



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview®
WWW.BROADVIEW.COM.CN

Csdn

技术凝聚实力 专业创新出版

博文视点 (www.broadview.com.cn) 资讯有限公司是电子工业出版社、CSDN.NET、《程序员》杂志联合打造的专业出版平台,博文视点致力于——IT专业图书出版,为IT专业人士提供真正专业、经典的好书。

请访问 www.dearbook.com.cn (第二书店) 购买优惠价格的博文视点经典图书。

请访问 www.broadview.com.cn (博文视点的服务平台) 了解更多更全面的出版信息; 您的投稿信息在这里将会得到迅速的反馈。

博文本版精品汇聚



加密与解密 (第三版)

段钢 编著

ISBN 978-7-121-06644-3

定价: 69.00元

畅销书升级版, 出版一月销售10000册。
看雪软件安全学院众多高手, 合力历时4年精心打造。



疯狂Java讲义

新东方IT培训广州中心

软件教学总监 李刚 编著

ISBN 978-7-121-06646-7

定价: 99.00元 (含光盘1张)

用案例驱动, 将知识点融入实际项目的开发。
代码注释非常详细, 几乎每两行代码就有一行注释。



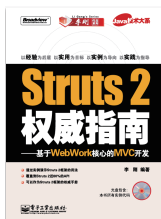
Windows驱动开发技术详解

张帆 等编著

ISBN 978-7-121-06846-1

定价: 65.00元 (含光盘1张)

原创经典, 威盛一线工程师倾力打造。
深入驱动核心, 剖析操作系统底层运行机制。



Struts 2权威指南

李刚 编著

ISBN 978-7-121-04853-1

定价: 79.00元 (含光盘1张)

可以作为Struts 2框架的权威手册。
通过实例演示Struts 2框架的用法。



你必须知道的.NET

王涛 著

ISBN 978-7-121-05891-2

定价: 69.80元

来自于微软MVP的最新技术心得和感悟。
将技术问题以生动易懂的语言展开, 层层深入, 以例说理。



Oracle数据库精讲与疑难解析

赵振平 编著

ISBN 978-7-121-06189-9

定价: 128.00元

754个故障重现, 件件源自工作的经验教训。
为专业人士提供的速查手册, 遇到故障不求人。



SOA原理·方法·实践

IBM资深架构师毛新生 主编

ISBN 978-7-121-04264-5

定价: 49.8元

SOA技术巅峰之作!
IBM中国开发中心技术经典呈现!



VC++深入详解

孙鑫 编著

ISBN 7-121-02530-2

定价: 89.00元 (含光盘1张)

IT培训专家孙鑫经典畅销力作!

博文视点资讯有限公司
电话: (010) 51260888 传真: (010) 51260888-802
E-mail: market@broadview.com.cn (市场)
editor@broadview.com.cn jsj@phei.com.cn (投稿)
通信地址: 北京市万寿路173信箱 北京博文视点资讯有限公司
邮编: 100036

电子工业出版社发行部
发行部: (010) 88254055
门市部: (010) 68279077 68211478
传真: (010) 88254050 88254060
通信地址: 北京市万寿路173信箱
邮编: 100036

博文视点 · IT出版旗舰品牌

《深入浅出 Linux 工具与编程》

读者交流区

尊敬的读者：

感谢您选择我们出版的图书，您的支持与信任是我们持续上升的动力。为了使您能通过本书更透彻地了解相关领域，更深入的学习相关技术，我们将特别为您提供一系列后续的服务，包括：

1. 提供本书的修订和升级内容、相关配套资料；
2. 本书作者的见面会信息或网络视频的沟通活动；
3. 相关领域的培训优惠等。

您可以任意选择以下四种方式之一与我们联系，我们都将记录和保存您的信息，并给您提供不定期的信息反馈。

1. 在线提交

登录 www.broadview.com.cn/13750，填写本书的读者调查表。

2. 电子邮件

您可以发邮件至 jsj@phei.com.cn 或 editor@broadview.com.cn。

3. 读者电话

您可以直接拨打我们的读者服务电话：**010-88254369**。

4. 信件

您可以写信至如下地址：北京万寿路173信箱博文视点，邮编：**100036**。

您还可以告诉我们更多有关您个人的情况，及您对本书的意见、评论等，内容可以包括：

- (1) 您的姓名、职业、您关注的领域、您的电话、E-mail地址或通信地址；
- (2) 您了解新书信息的途径、影响您购买图书的因素；
- (3) 您对本书的意见、您读过的同领域的图书、您还希望增加的图书、您希望参加的培训等。

如果您在后期想停止接收后续资讯，只需编写邮件“退订+需退订的邮箱地址”发送至邮箱：

market@broadview.com.cn 即可取消服务。

同时，我们非常欢迎您为本书撰写书评，将您的切身感受变成文字与广大书友共享。我们将挑选特别优秀的作品转载在我们的网站（www.broadview.com.cn）上，或推荐至CSDN.NET等专业网站上发表，被发表的书评的作者将获得价值50元的博文视点图书奖励。

更多信息，请关注博文视点官方微博：<http://t.sina.com.cn/broadviewbj>。

我们期待您的消息！

博文视点愿与所有爱书的人一起，共同学习，共同进步！

通信地址：北京万寿路 173 信箱 博文视点（100036）

电话：010-51260888

E-mail：jsj@phei.com.cn，editor@broadview.com.cn

www.phei.com.cn
www.broadview.com.cn

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：(010) 88254396; (010) 88258888

传 真：(010) 88254397

E-mail: dbqq@phei.com.cn

通信地址：北京市万寿路 173 信箱

电子工业出版社总编办公室

邮 编：100036

深入浅出Linux工具与编程

- 这是一本有思想、有内容的图书
- 这是一本回答学什么、怎么学的图书
- 这是一本大学毕业生一次阅读、终生受益的图书
- 这是一本集实用性、典型性、模仿性案例的图书
- 这是一本很通俗易懂的图书
- 这是一本帮你突破技术玻璃纸、一通百通的图书
- 这是一本包含作者从业十年心得经验的图书
- 这是一本零起点的Linux专家速成培训教程

本书系统地论述了Linux工具与编程的相关知识。全书内容可分为两部分：Linux知识的初级部分和高级部分。其中初级部分包括Linux操作系统介绍、Linux命令说明、Linux常见实用工具（正则表达式、find、sed、awk）、Shell编程、Linux C语言程序设计、Linux C语言开发工具（vi与vim编辑器、gcc、Makefile和gdb）；高级部分包括Linux进程编程（Linux进程、Linux线程、管道与信号、消息队列、信号量和共享内存）、Linux文件编程、网络编程和XML编程。

本书初级部分适合高等院校相关专业学生和Linux爱好者学习，高级部分适合Linux行业资深从业人员学习。



策划编辑：张春雨
责任编辑：李利健 贾莉
封面设计：侯士卿

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

上架建议：Linux编程

ISBN 978-7-121-13750-1



定价：79.00元